Model-driven Engineering for Early QoS Validation of Component-based Software Systems

James H. Hill and Aniruddha Gokhale
Vanderbilt University, Nashville, TN, USA
Email: {hillj, gokhale}@dre.vanderbilt.edu

Abstract— **Model-driven engineering (MDE) techniques are increasingly being used to address many of the development and operational lifecycle concerns of large-scale component-based systems. One such concern that is growing in importance, but lacking significant research is the validation of quality-of-service (QoS) properties of component-based systems throughout their development lifecycle. In the current state of the art, large-scale component-based systems have to wait until system integration time to perform in-depth QoS testing, which can be too late and detrimental to project schedules and costs. This paper describes our novel MDE-based solution to address this challenge. At the core of our solution approach are (1) a set of domain-specific modeling languages that allow us to mimic component "business logic," and (2) a generative programming framework that synthesizes configuration files for system simulation. A particular thrust of this paper is describing the syntax and semantics of the component behavior modeling language, which are based on the Input/Output automaton formalism.**

Our experience shows that our techniques enable developers of large-scale component-based systems to perform early QoS evaluation from development time to production time. Moreover, our experience shows we are able to provide an environment where developers can rapidly test ideas and methodologies largely alleviating the need to expend effort, money, and time implementing them.

Index Terms— **model-driven system engineering, continuous QoS validation, simulation, timed input/output automata**

I. Introduction

Model-driven engineering (MDE) [1] techniques are increasingly being used to address many of the development and operational lifecycle complexities of large-scale component-based systems. Although there have been many advances in MDE techniques for large-scale component-based systems, MDE techniques to date have focused primarily on (a) structural issues of system development, such as component assembly, packaging, configuration and deployment [2]–[4], and (b) functional and behavioral issues, such as model checking for functional correctness (*e.g.*, Bogor [5] and Cadena [6]) or runtime validation of performance (*i.e.*, running empirical benchmarks at integration time to validate performance).

Although MDE tools continue to raise the level of abstraction of component-based software systems and address many of their complexities, there remains a major gap in evaluating system quality of service (QoS), *e.g.*, performance and reliability, at different phases of development, which would enable design flaws to be rectified earlier in the development lifecycle. This impediment is due primarily to the "serialized phasing" [1] nature of the development lifecycle wherein the system is developed in layers (*e.g.*, first the components at the infrastructure layer(s) and then the application layer(s)).

Throughout the development of each layer, the business-logic encapsulated within individual components is usually thoroughly tested for both functional correctness (*i.e.*, whether it performs the expected operations) and performance correctness (*i.e.*, whether operations are performed with the expected QoS). Due to the composite nature of large-scale component-based software systems, complete system QoS validation can proceed only when all the system components are available and deployed in the runtime infrastructure [7], [8]. Moreover, waiting too late in the development lifecycle (*e.g.*, integration time when all components are available) to resolve any performance problems can be too costly to resolve. It is clear that system engineers need proper tools to help address QoS validation not only at integration and production time, but at development time before performance problems become too "hard" to locate and resolve.

In our previous research [9]–[12], we showed how emulating system components for QoS validation while the "real" components are being developed enables developers to perform QoS validation throughout the entire development lifecycle. At the core of our solution are two domain-specific modeling languages (DSMLs) [13] named the Component Behavior Modeling Language (CBML) and the Workload Modeling Language (WML), which allow developers to define the behavior and workload, respectively, of component-based systems at a higher-level of abstraction from that provided by third generation programming languages [12]. In addition, we implemented a reusable generative programming framework for CBML and WML that allows us to generate configuration files and/or source code for different use cases [14]. In this paper, we extend our previous research efforts and focus on the formal definition of a component in terms of CBML. Moreover, we illustrate how we are able to generate configuration files for validating components via simulation at design-time using a technique called *semantic anchoring* [15]–[17], which uses well-defined transformations to map a DSML to an existing formal language (*e.g.*, input/output automata [18] and timed-automata [19]) to validate formally define the semantics of the DSML.

Paper Organization. The remainder of this paper is organized as follows: Section II introduces a motivating example we use to describe the challenges in realizing a solution

This work was supported by a Raytheon IRAD grant

for early QoS evaluation; Section III describes the structure and functionality of our DSML for simulating component behavior; Section IV explains how we integrate our DSML with existing structural DSMLs to associate behavior models with structural models; Section V explains how we use semantic anchoring to generate configuration files for simulation; Section VI compares our work with related research; and Section VII presents concluding remarks.

II. Motivating Example

This section describes the challenges in developing a solution that addresses the need for early QoS evaluation of component-based systems developed using serialized phasing processes. We use a motivating example to highlight these challenges.

A. A Distributed Stockbroker Application Case Study

We use a representative example taken from the financial domain [20] as a motivating example to illustrate the serialized phasing problem and how our research artifacts described in this paper enable us to provide early QoS validation. Our case study is called the Distributed Stockbroker Application (DSA), which is an online web application for viewing stock information.

Figure 1 shows a high-level representation of the DSA and its communication flows between components. The DSA is composed of six different components. The *Naming Service* component allows client applications to locate the *Gateway Component* for the application. The location (*i.e.*, the binding IP address and port number) of the naming service component is therefore persistent. The *Gateway Component* serves as the entrance to the stock application, which all clients must pass through. The *Gateway Component* accepts the username and password of the user and sends it to the *Identity Manager* component. The *Identity Manager* component is responsible for verifying the username and password, and initializing the correct QoS policies based on user type. Once the access is granted to the client, it is given direct access to a *Stock Component*. The *Stock Component* is created on-demand and initialized with the correct QoS specified by the *Identity Manager*. The *Stock Component* interacts with a MySQL database that contains the stock information. Lastly, all components in the system – both application and infrastructure – log their activities to a *Logging Component*.

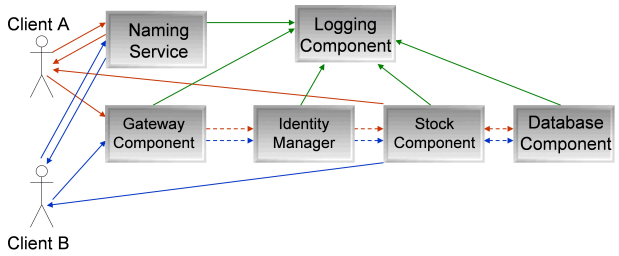

Fig. 1. High-level structural composition of the Distributed Stock Application.

The DSA has two user classes: *Basic* and *Gold*. *Gold* users are persons who use the service frequently, whereas *Basic* users use the service infrequently. Table I provides the projected usage pattern and desired response times (*i.e.*, QoS) of each user for the DSA. Due to the serialized-phasing development process, the underlying infrastructure of the DSA (*i.e.*, all the components illustrated in Figure 1) may complete their development at different time scales. Evaluating system design decisions on the target architecture to understand and evaluate system QoS, therefore, has to wait until all the "real" components are available.

TABLE I
Predicted usage pattern of the Distributed Stock Application based on user type.

Type	Percentage	Response Time (msec)
Basic (Client A)	65%	300
Gold (Client B)	35%	150

The application components of DSA are implemented as Lightweight CORBA Component Model (CCM) [21] components. The target architecture comprises three hosts for deploying all its components. Lastly, the software platform version is Fedora Core 4 using ACE+TAO+CIAO 5.1 middleware platform available at `www.dre.vanderbilt.edu`.

B. Impediments to Overcoming the Serialized-phasing Barrier

To achieve the vision of early QoS validation in the presence of serialized phasing, such as in the case of the DSA case study, the proposed solution must address the following challenges:

- **Challenge 1: Capture business logic** – The components must resemble their counterparts in both supported interfaces and behavior. For emulation, the target environment should allow seamless replacement of faux components with real components as they become available. For simulation, however, seamless replacement is not applicable. The configuration files for simulation must define elements (*e.g.*, inputs, outputs, and transitions) that resemble their real counterpart to preserve similarity and contextual representation.
  In the context of the DSA, emulated components should be used to evaluate QoS at early stages of development, and as the "real" components are available they should replace the emulated components to achieve more accurate QoS metrics. Likewise, the simulated components should be used to verify properties such as functional correctness and reachability.
- **Challenge 2: Realistic mapping of behavior** – The behavior specification should operate at a high-level of abstraction (*i.e.*, at the application level) and map to realistic operations (*e.g.*, memory allocations and deallocations, file operations, or database transactions).
  For example, in the DSA the high-level database behavior should "realistically" query a database for stock information when using emulation. In the context of simulation, the behavior should map to well-defined elements of the underlying formal language that represent querying a database.

- **Challenge 3: Technology independence** – The behavior specification should not be tightly coupled to a programming language, middleware platform, hardware technology, and MDE tool.
  In the context of the DSA, if we wanted to evaluate the system on CCM or Microsoft .NET [22], or use multiple modeling tools [23], [24], then we should be able to reuse the same concepts and models. Likewise, if we wanted to simulate the DSA under different tools such as Tempo (`www.veromodo.com`) or UPPAAL (`www.uppaal.com`), we should be able to reuse the same models.

The remainder of this paper describes our solution to resolve these challenges.

III. Domain-specific Modeling Languages for Early QoS Validation

Addressing the challenges of continuous QoS evaluation in the face of serialized phasing requires mechanisms to mimic application component behavior. This section describes our R&D on a DSML named the Component Behavior Modeling Language (CBML). CBML is a DSML for capturing the behavior of a component and is primarily used to generate configuration files for simulation tools. The remainder of this section discusses CBML in detail explaining how CBML helps resolve Challenges 1 and 2 discussed in Section II-B in the context of the case study described in Section II-A.

A. The Component Behavioral Modeling Language

Any mechanism that mimics component behavior must incorporate the design principles and semantics of component architectures. In such architectures, systems are composed of components that react to method invocations and events received on their input ports. This "reaction" causes a sequence of activities that can be defined by a series of states and transitions. Although the range of activities performed in the course of a component's execution can vary broadly, they can be divided into two distinct operational classes: *internal* and *communication*.

Internal operations are those not observable from outside a component (*e.g.*, memory allocations/deallocations and database transactions executed by the database component in the DSA case study). Communication operations are representative of sending/receiving an event to/from another component (*e.g.*, input and output events transmitted between each of the components in the DSA case study).

When trying to emulate a component's behavior (*i.e.*, addressing Challenge 1 in Section II-B), it is desirable to capture it as close as possible to its real counterpart using combinations of internal and communication operations. It is also desirable to represent the behavior based on a formal mathematical foundation because it will (1) facilitate transformation of existing models between different formal behavioral languages (*e.g.*, timed-automata, StateCharts [25] and PetriNets [26]), and (2) assist in proving any formal properties of the system (*e.g.*, correctness and stability). Likewise, it will also facilitate reverse transformations (*i.e.*, from models in other languages to models of this language). We believe that lack of formal semantics can limit the capabilities and scope of such a behavioral modeling language. At the same time, it should not be dependent on any programming language and software/hardware platform, and be as general purpose as possible.

Based on the desired functionality for modeling component behavior, we have developed the Component Behavioral Modeling Language (CBML). CBML is a DSML based on the mathematical formalism of input/output (I/O) automata [18] (details of I/O automata are beyond the scope of this paper). We chose I/O automata as its basis because, analogous to component behavior, I/O automata is ideal for asynchronous and reactive systems. We developed CBML in the Generic Modeling Environment (GME) [23], which is a metamodeling environment that allows the creation of DSMLs and its models. CBML, however, is not coupled to GME since it can be ported to any MDE tool that supports metamodel specification (*e.g.*, Eclipse Modeling Framework (EMF) [27], Generic Eclipse Modeling System (GEMS) [24], or Microsoft DSML tools [28]). Developers use CBML to capture component behavior at a high-level of abstraction and use model interpreters to generate configuration and source files for backend emulation (see Section IV) and simulation (see Section V) tools.

1) Structure of CBML: As explained in Section III-A, we developed CBML based on the mathematical formalism called I/O automata [18]. We defined CBML so that it has the necessary subset of elements from I/O automata that will preserve its formal semantics. Users of CBML do not need prior knowledge of I/O automata to use CBML. Keeping that in mind, we formally define the behavioral model $BM = (V, S, \Theta, I, O, A, T, E)$ of a component in CBML as:

- A set V of *internal variables*.
- A set $S \subseteq val(V)$ of *states* where $val(V)$ is the value of the internal variables at any given point in time, or the current state of the component.
- A nonempty set $\Theta \subseteq S$ of *start states*.
- A set I of *input actions*, which are events received from an external source, *e.g.*, a connected component.
- A set O of *output actions*, which are events sent to an external destination, *e.g.*, a connected component.
- A set A of *actions*, which are events (or actions) visible only to the component hosting the behavior, *i.e.*, internal operations.

Figure 2 highlights each of these elements in BM as their representative artifacts in CBML.

Fig. 2. Primary elements for constructing behavioral models in CBML.

In order to construct valid behavioral models in CBML, developers must specify a sequential flow between different actions $\Omega = (I \cup O \cup A)$ where Ω is a disjoint union and states S. Behavioral models BM, therefore, contain two sets of functions that allow developers to define sequential flows as follows:

- A set T of *transitions* such that given $\Delta \in T$ and $s \in S$:

$$\Delta(s) \rightarrow \alpha, \tag{1}$$

  where $\alpha \in (A \cup O)$.
- A set E of *effects* such that given $\Gamma \in E$ and $a \in \Omega$:

$$\Gamma(a) \rightarrow s, \tag{2}$$

  where $s \in S$.

Figure 3 shows the complete realization of BM using the respective CBML artifacts illustrated in Figure 2, Equation (1) and Equation (2) in the context of the DSA database component. In CBML, all behavioral specification begins with an *Input Action* element. Each *Input Action* in the behavior model is connected to an initial *State* element. The remainder of the behavioral specification is defined by a sequence of *Action* to *State* transitions. For example, the behavioral model for the database component in Figure 3 illustrates that an input action causes a query for stock information.

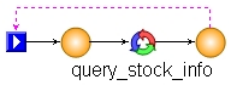

Fig. 3. Example CBML behavioral model in GME.

To specify the end of a behavior sequence in the modeling realm, a *Finish* connection (*i.e.*, the dashed line) is used to connect the final *State* to the starting *Input Action*. We require this connection because we allow sharing of behavioral sequences to simplify modeling (illustrated in Figure 4). For example, the DSA has two type of users who have the same behavior. It is possible to model each person's input to the database component (or any component) separately but share the same behavior as illustrated in Figure 4.[1] The explicit finish connections therefore help resolve ambiguity when determining where each user type's behavior terminates.

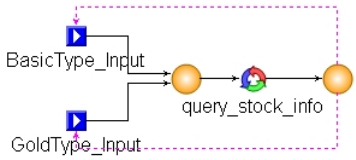

Fig. 4. Example of sharing behavior in CBML.

During the interpretation process of CBML, we treat shared behavioral sequences as separate sequential flows to preserve the validity of Equation (1) and Equation (2). Figure 5 illustrates how Figure 4 is handled during the interpretation process to generate either emulation or simulation files. As shown in Figure 5, there are now 4 different states and 2 different actions, thus preserving the validity of Equation (1) and Equation (2).

[1]Shared behavior is a modeling optimization we allow to help reduce the size of constructed models because automata-based models are affected by state-space explosion as they grow in the number of elements.

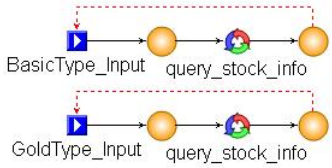

Fig. 5. View of shared behavior in CBML from the interpretation perspective.

Specifying output actions in CBML: CBML defines behavior as an input action that causes a series of "internal" operations and results in a set of output actions, if any. Based on the definitions of a transition Δ from Equation (1), it is clear that output actions O have the same *modeling* semantics as actions A.

Figure 6 illustrates an example behavioral model with output actions, which are represented by the three rightmost squares labeled `basic_response`, `gold_response` and `log_status` for the database component in the DSA. After the component completes its query to the database for stock information, it outputs information back to the requester, and outputs a status message to the external logging component.

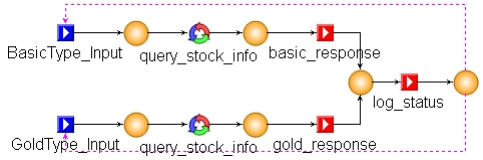

Fig. 6. Example CBML behavioral model with output actions.

Preconditions, postconditions, and variables in CBML: CBML allows users to define variables V in behavioral models to preserve information that represents the current state of the component, $val(V)$. Preconditions, which are associated with transitions Δ, operate on the variables to enable and/or disable the execution of individual transitions. Likewise, postconditions, which are associated with effects Γ, modify the values of variables to change the current state of the component, or system. Formally, preconditions and postconditions are defined as follows:

- For preconditions:

$$\Delta(s) \leftrightarrow pre(val(V)) \tag{3}$$

  where $pre(X)$ determines if the current value of X is `true`.
- Upon execution of effect Γ associated with action a

$$post(a) \rightarrow val(V)', \tag{4}$$

  where $a \in \Omega$ and $val(V)'$ is the new state of the system such that $S \subseteq val(V)'$.

As illustrated in Figure 7 in the context of CBML, a variable is represented by the element with the star image. Users use variables in their behavioral model by referencing them in the preconditions and postconditions of the transition (*i.e.*, connection from a state to an action), and effect (*i.e.*, connection from an action to a state) connections, respectively.

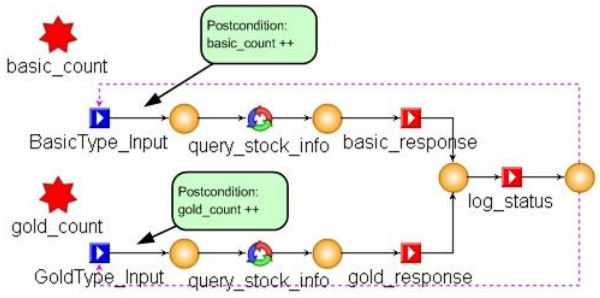

Fig. 7. Example CBML behavioral model with variables.

This allows developers to create more "realistic" behavioral models, such as counting the number of users of each type executing queries on the database and/or guarding a workload until the system reaches a certain state.

Domain-specific extensions in CBML: Some input events that are critical in the domain of component-based systems (*e.g.*, lifecycle events such as *activation* and *passivation* or monitoring notification events such as degradation of QoS) are not first class entities in I/O automata. I/O automata does not distinguish between these kinds of events because it is a general-purpose language that is not tied to any particular domain (*e.g.*, component-based systems). We therefore extended I/O automata (without affecting its formal semantics) in CBML to capture this aspect of component behavior more expressively as discussed below and illustrated in Figure 8:

- **Environment Events**, $E \subseteq I$, represent input actions to a component that are triggered by the hosting system rather than another component (*e.g.*, lifecycle events from the hosting container or fault-tolerance notifications to serialize the state of a component).
- **Periodic Events**, $P \subseteq I$, represent input actions from the hosting environment that occur periodically (*e.g.*, setting/receiving a timeout event to periodically transmit status updates). We also allow a probability to be associated with periodic events to provide non-deterministic behavior.

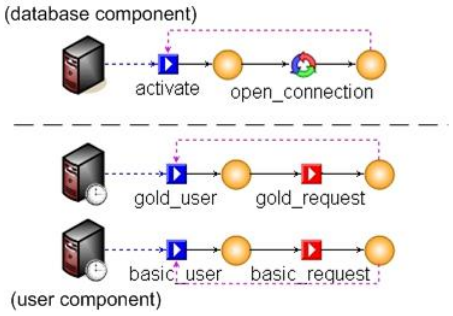

Fig. 8. CBML's domain-specific extensions to I/O automata

In the context of the DSA, when the database component is activated it creates an initial connection to the database (illustrated in Figure 8). Likewise, we can use periodic events to model the behavior of each user type by associating each one with correct probability (*e.g.*, 0.35 and 0.65 for Gold and Basic type, respectively) and sequencing it with an output event within a "user" component (also illustrated in Figure 8).

Usability extensions in CBML: One of the main goals of defining behavior at a high-level of abstraction is simplicity and ease of use. If the size of the behavioral model is "huge" and CBML adheres strictly to its current representation of I/O automata, its ease of use is compromised because one of the major drawbacks of many automata languages is scalability [25]. To address this issue we defined the following usability extensions, which does not violate the definition of BM in CBML:

- **Composite Action**, $CA \in A$, is a modeling element that allows developers to create reusable behavior workflows that can help reduce the amount of clutter in the model. A composite action has the same definition as BM, however, we defined a constraint that requires composite actions to contain only a single input action, *i.e.*, $|I| = 1$. This is necessary because composite actions encapsulate a single, reusable behavior workflow, and not multiple behavior workflows.
- **Log Action** is an attribute of an *Action* element that determines if the action should be logged. The semantics of "logged" are dependent on how the model is interpreted. For example, a modeler might choose to log "network send" actions and not "memory allocation" actions.

To address the usability concerns in the modeling aspect, we also developed a GME add-on that assists users in creating models rapidly by auto-generating required elements (*e.g.*, states) and connections depending upon the context. Although this feature is GME-specific, most MDE tools provide support for implementing features that help improve user experience [29].

B. Supporting Timing Semantics in CBML

Input/Output (I/O) automata is ideal for modeling asynchronous, reactive systems, such as large-scale component-based DRE systems. When trying to evaluate QoS, however, I/O automata lacks several aspects, such as timing, that would allow developers to verify QoS properties about components, and the system (*e.g.*, end-to-end deadlines, expected execution time, etc.).

To address this limitation, Timed Input/Output Automata (TIOA) [30] was defined as an extension I/O automata to support timing aspects. TIOA has the same formal semantics as I/O automata, but it is extended to support both discrete and continuous variables. The continuous variables (*e.g.*, a clock or temperature) define how the state of the system changes with respect to time.

Because CBML was originally based on the semantics of I/O automata, it also lacked the same properties that would allow developers to verify QoS properties from a simulation standpoint. We, therefore, extended CBML to support the notion of timing to be consistent with TIOA. In CBML, timing is defined by the following equation:

$$clock' = clock + time(a) \tag{5}$$

where $clock$ is the current timing variable for the component, $time(a)$ is the execution duration of $a \in A$, and $clock'$ is the new clock time after completing a. In CBML, we only associate timing with internal actions A because we, currently, make the assumption that all input I and output O actions are instantaneous.

IV. Integrating Behavioral and Structural DSMLs

In Section III, we described a behavioral DSML named CBML and illustrated how it allows us to capture the behavior (Challenge 1 of Section II-B) and map the behavior to realistic operations (Challenge 2 of Section II-B). Although CBML allows us to capture the behavior of a component, the models are insufficient to generate simulation code directly without knowing the structural composition of the system and its components for QoS validation since the latter determines the end-to-end workflows.

We, therefore, integrated CBML with the Platform Independent Component Model Language (PICML) [31] because PICML captures the structural aspects of a system and its components. Moreover, since both PICML and CBML provide platform and programming language independent modeling capabilities, their integration and model interpretations provide a technology independent approach to continuous QoS evaluation (Challenge 3 in Section II-B).

Although we chose PICML as the structural DSML to integrate CBML, the concepts presented in Section IV-A can be applied to any structural DSML provided that it clearly differentiates between input and output ports of a component. The remainder of this section discusses integration of CBML with existing languages (*e.g.*, PICML in CoSMIC), and how our approach to generating simulation logic for components that mimics their real capabilities is decoupled from the underlying platform and programming language technology.

A. Integrating CBML and PICML

Domain-specific modeling languages (DSMLs), such as PICML, allow developers to model different ports of a component (*e.g.*, facet/receptacles and event sources/sinks). The facets/event sinks represent inputs to a component, while receptacles/event sources represent outputs from a component. Formally, a basic component $C = (M, N)$ from the structural aspect can be defined as:

- A set M of input ports for receiving events from external sources, *e.g.*, connected components.
- A set N of output ports for sending evens to external destinations, *e.g.*, connected components.

Structural DSMLs, however, capture structural input/output (I/O) elements without any correlating behavior (*i.e.*, there is no clean representation to associate the I/O elements of structural models with the I/O actions in behavioral models). We, therefore, extended the structural definition of a component $C = (M, N, \Phi, \Psi)$ to define a set of functions that enable developers to connect the I/O elements in the structural model with corresponding I/O elements in the behavioral model BM (see Section III-A.1) based on the following equations:

- Let $m \in M$, $i \in I$ and $\phi \in \Phi$, then
$$\phi(m) \rightarrow i. \tag{6}$$
- Let $n \in N$, $o \in O$ and $\psi \in \Psi$, then
$$\psi(o) \rightarrow n. \tag{7}$$

Figure 9 illustrates how structural DSMLs (*e.g.*, PICML) that define components that have I/O ports and behavioral DSMLs (*e.g.*, CBML) that have I/O actions can be integrated by having the structural DSML "contain" the behavioral DSML and applying Equation (6) to the structural DSML and Equation (7) to the behavioral DSML.

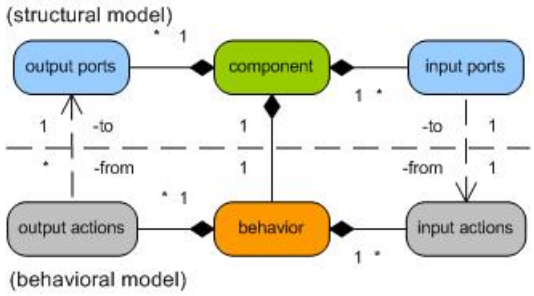

Fig. 9. Conceptual model of integrating behavioral and structural DSMLs.

In the modeling realm, we require a component to contain the behavior. Additionally, we define a modeling connection between the input port and input action to implement Equation (6), but require that the name of the output action match the name of the corresponding output port to implement Equation (7). We made this design decision because explicitly defining a connection between an output action and port will clutter the model since there is a many-to-one mapping between an output action and an output port.

To further illustrate this concept, Figure 10 shows the realization of integrating a behavioral and structural DSML. The outer rectangle of Figure 10 illustrates the PICML model of the database component. The inner rectangle highlights the same database component with CBML from Figure 7 integrated into PICML, thus allowing us to model the same behavior exemplified in Section III with its respective structure (*e.g.*, interface and attributes).

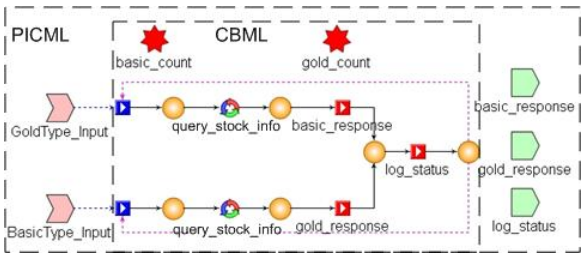

Fig. 10. Realization of integrating CBML and WML with PICML in CoSMIC.

V. Semantic Anchoring for Simulation of Component-based Software Systems

In Section IV, we discussed how we integrated CBML with structural DSMLs (*e.g.*, PICML) to associate behavior models

with structural models. In this section, we discuss how we use semantic anchoring [15]–[17] to generate configuration files for simulation tools based on timed I/O automata (TIOA). We limit our discussion to the generation of configuration files for individual components, and not the entire system (*e.g.*, nodes, communication channels, and etc.) because it is outside the scope of this paper.

A. Brief Overview of Semantic Anchoring

One of the main benefits of a DSML is its ability to allow developers to work with artifacts that are familiar to their domain. Although a DSML can help raise the level of abstraction – and simplify the development process by automating tedious and error-prone tasks – many DSMLs lack methods for proving their validity through formal specification of their semantics. Because it can be "hard" to formally define the semantics of a DSML in ways similar to formal mathematical languages such as I/O automata, Timed Automata, and Statecharts, it is becoming common practice to leverage semantic anchoring as a method to formally define the semantics of a DSML.

In semantic anchoring, developers rely on well-defined transformations that map elements of the DSML in question to elements of an existing formal language. This then alleviates the necessity to formally define the semantics of a DSML because if the transformation functions are well-defined and the target language is semantically valid, then one can argue that the semantics of DSML in question is formally defined in the context of the target language. The remainder of this section discusses how we use semantic anchoring to map CBML to TIOA.

B. Transforming CBML into Timed I/O Automata

When we originally designed CBML, we based its definition on aspects from I/O automata because I/O automata possessed many of the same characteristics of components. In order to validate QoS from a simulation standpoint – as opposed to an emulation standpoint – we extended CBML to support timing so it would be consistent with TIOA. This would permit us to start understanding QoS properties such as end-to-end deadlines, service rates, and expected execution times, from simulation perspective.

In TIOA, an automaton $\mathcal{A} = (\beta, I, O)$ is a tuple where

- $\beta = (X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$ is a timed automata.
- I and O partition E into *input* and *output* actions, respectively.

We do not present the complete definition of B and its properties in the paper, and encourage the reader to see [30] for more details.

In order to leverage TIOA for semantic anchoring, we must first define a set of transformations that map CBML to TIOA. It is obvious that many of the elements in the definition of BM, which is used to formally define CBML, already occur in $\mathcal{A}$. Therefore, when we use the following transformation function:

$$trans(X_{BM}) \rightarrow Y_{\mathcal{A}}, \tag{8}$$

where X is an element in the definition of BM that is being transformed into element Y in the definition of $\mathcal{A}$, we define the following transformations:

$$trans(V_{BM}) \rightarrow X_{\mathcal{A}}, \tag{9}$$

$$trans(S_{BM}) \rightarrow Q_{\mathcal{A}}, \tag{10}$$

$$trans(\Theta_{BM}) \rightarrow \Theta_{\mathcal{A}}, \tag{11}$$

$$trans(I_{BM}) \rightarrow I_{\mathcal{A}}, \tag{12}$$

$$trans(O_{BM}) \rightarrow O_{\mathcal{A}}, \tag{13}$$

$$trans(A_{BM}) \rightarrow H_{\mathcal{A}}, \tag{14}$$

$$trans(T_{BM}) \rightarrow \mathcal{D}_{\mathcal{A}}, \tag{15}$$

$$trans(E_{BM}) \rightarrow s_{\mathcal{A}}, \tag{16}$$

where $s_{\mathcal{A}} \in Q_{\mathcal{A}}$.

To further illustrate the transformation, we have applied the transformation functions to a simplified version of the database component in the DSA illustrated in Figure 11. The simplified version of the database component contains an input action named `BasicType_Input` and an internal action named (`query_stock_info`). It also contains a single output action named (`send_result`). When we apply Equation (9) - Equation (16) to the CBML model in Figure 11, we produce the TIOA configuration file presented in Listing 1.

Fig. 11. Simplified version of the database component in the DSA.

```
automaton DatabaseComponent (M: type)
  signature
    input BasicType_Input (m: M)
    internal handle_BasicType_Input
    internal query_stock_info
    output send_result (n: N)

  states
    next: Int := 1;
    queue_BasicType_Input : Seq[M] := {};
    clock : Int := 0;

  transitions
    input BasicType_Input (m)
      eff queue_BasicType_Input :=
        queue_BasicType_Input |- m;

    internal handle_BasicType_Input
      pre next = 1 /\
          queue_BasicType_Input ~= {};
      eff queue_BasicType_Input =
```

```
        tail (queue_BasicType_Input);
        next := 2;

    internal query_stock_info
      pre next = 2;
      eff next := 3; clock := clock + 10;

    output send_result
      pre next = 3;
      eff next := 1;

    trajectories
      trajdef thread
        evolve d(clock) = 1;
```

Listing 1. Timed Input/Output Automaton configuration file for the simplified database component.

As shown in Listing 1, the database component is converted to a single TIOA named `DatabaseComponent` that has a generic message type for receiving events. Each of the actions (*i.e.*, input, output, and internal) are converted to their equivalent TIOA element based on Equation (12), Equation (13) and Equation (14), respectively. Lastly, there is implicit corresponding internal action named `handle_BasicInput_Event` that is responsible for triggering the behavior sequence when an event is received on `BasicInput_Event` and placed in the corresponding event queue.

The `DatabaseComponent` automaton also contains three variables that hold the current state of the component (*i.e.*, $val(V)$). The `next` variable – which *every* component defines – determines what action to execute next in the behavior sequence since CBML sequences its behavior workflow. The `queue_BasicType_Input` is the event queue that stores events received on `BasicType_Input`. Each input action in CBML that is associated with an input event channel of a component *always* has an associated event queue. The `clock` variable is a continuous variable that represents time of execution. Its evolution is defined in the `trajectories` section of the automaton. Lastly, although the database component does not contain any explicit CBML variables, if the behavior model has any CBML variables, they are defined the `states` section of the automaton.

Each of the CBML actions (*i.e.*, input, output, and internal) are converted to their respective TIOA elements. Because CBML forces a sequencing of the operations, we also defined TIOA preconditions (`pre` statements) and effects (`eff` statements) that will enforce this sequencing. As highlighted in Listing 1, `handle_BasicInput_Event` is not enabled until `BasicInput_Event` has fired (*i.e.*, successfully executed its effects to change the automaton's state). Likewise, `query_stock_info` cannot fire until `handle_BasicInput_Event` fires and `send_result` cannot fire until `query_stock_info` fires. Because internal actions have a timing aspect associated with them, the effect of firing an internal actions also modifies the `clock` variable by the specified time, *i.e.*, Equation (5), to simulate the execution duration of the associated action. Lastly, it is clear that we only allow a single event to be active per behavior sequence, and not per component; however, we do not show this in the presented example.

C. Simulating Timed I/O Automata Models

Once CBML models are converted to TIOA models, developers can use them to define simulations that check different properties of individual components. The TIOA components we generate do not define complete simulations of the components because from a simulation standpoint, we do not know how developers want to follow different trajectories defined in the components. Instead, we generate the minimal sized component that allows developers to combine them with other TIOA models that define more trajectories, or simulations threads, to exercise the components. Moreover, since the models are converted to TIOA, developers can also leverage tools such as Tempo (`www.veromodo.com`), which has tools and plug-ins to convert TIOA models to Timed Automata models, and other models types for theorem proving tools, thus satisfying Challenge 3 in Section II-B.

VI. Related Work

This section compares our work on using DSMLs for modeling system behavior to evaluate component-based systems with related research efforts.

A. Formal Languages

Statecharts [25] gained widespread usage when they were integrated with the STATEMATE [33] modeling tool, and since then a variant has became part of UML (*i.e.*, UML Statecharts) [34]. Similar to CBML, statecharts can be used to describe behavior of large complex systems. CBML extends Statecharts by clearly separating component behavior from workload. The generative techniques associated with variants of statecharts are targeted towards simulation and runtime verification [35], [36]. Our generative techniques can be extended to simulation and runtime verification tools [37] as well. Furthermore, our generative techniques and concepts are not tied to a specific technology or tool, whereas the technique presented in [38] et al., is bound to a specific tool.

The Abstract State Machine Language (AsmL) [39] developed at Microsoft Research is an executable specification language based on the theory of Abstract State Machines. AsmL is useful when developers need precise, non-ambiguous methods to specify a system, either software or hardware. AsmL, however, is not a graphical modeling language like CBML. Furthermore, users of CBML operate at a high-level and do not require in-depth knowledge of the underlying formalism, whereas AsmL requires developers to have some understanding of abstract state machines and programming formalisms, which can restrict its applicability (*e.g.*, for system testers who have no knowledge of complex formalisms or programming).

B. Process Modeling Languages

WinFX Workflow [42] is a process modeling language developed by Microsoft et al., which is a part of the Windows Workflow Foundation. Similar to CBML, WinFX allows developers to express workflows but it is coupled with workload. WinFX also facilitates code generation, but is confined to the

Microsoft .NET framework whereas our generative programming technique is technology and tool independent and can be applied to multiple middleware platforms including Microsoft .NET.

Java Workflow Tooling (JWT) [43] is a process modeling language for J2EE applications, however, it is still under development. Similar to CBML, JWT will allow developers to model the process of J2EE applications at a higher-level of abstraction using artifacts that are similar to their domain. CBML extends JWTs efforts because unlike JWT, CBML is not coupled to a specific programming language or technology. Moreover, CBML has defined formal semantics that allow it to be used either for emulation or simulation purposes.

The Business Process Modeling Notation (BPMN) [44] is a standard developed by Business Process Management Initiative (BPMI) that allows developers draw business processes in the form of workflows. Similar to CBML, BPMN is not platform, programming language, or technology dependent. CBML, however, extends BPMN by formally defining its semantics based on semantic anchoring so that it can leverage existing tools and techniques designed to operate on the formal language to which it is transformed.

VII. Concluding Remarks

This paper described a model-driven generative programming approach to address the challenges of evaluating component-based system QoS throughout the development lifecycle instead of delaying it to integration time. Our approach defined a modeling languages named CBML that captures the behavior of application components at a high-level. We then integrated CBML with PICML, which models structural properties of applications. Lastly, we used model interpreters to map the behavior specifications to operations that leverage existing simulation frameworks.

This approach allows for continuous integration and QoS validation of the system because as more is learned about the components, the behavior can be refined and regenerated for simulation. We expect the results of real versus simulated components to match provided the behavioral models of the simulated components closely approximate the real component's behavior.

A. Lessons Learned

Model-driven engineering comprising the use of DSMLs and generative programming provides an effective solution to address the challenges facing development lifecycles of next generation, large-scale software systems. Several challenges were encountered during the development of CBML and several challenges remain to be resolved. Our experience developing and using the MDE framework described in this paper suggests the following benefits:

- Using a DSML based on a mathematical formalism to define behavior of components helps in specifying unambiguous behavior when generating configuration files for simulation.
- Leveraging semantic anchoring simplifies validating a DSML because we only had to focus on defining the transformation to the underlying formal language as opposed to formally proving the validity of the DSML.

B. Future Work

Although our approach of integrating a behavior modeling language with a structural language has many benefits and addresses many challenges of the "serialized-phasing" process, there is also room for improvement and future work:

- Despite the ability to capture behavior of a component and its state, data flow of a component can only be defined based on state variables. In real world, properties of input actions (*e.g.*, event values) can affect the flow of execution in a real component. We, therefore, need to extend CBML with a simple programming language that will allow developers to use such properties when defining behavior.
- Currently, we make the assumption that only a single event can be active on a behavior sequence; and multiple events can be active as long as each event represents a separate behavior sequence. Components, however, can have multiple events active in a behavior sequence depending on the number of threads active for an input event. We, therefore, need to extend our simulation efforts to support the concept of multi-threaded input events.

By providing these extensions to our MDE approach, we will be able to continue addressing many of the challenges component-based system developers experience when they face time-to-market and product quality pressures.

References

[1] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[2] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. S. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt, "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*, 2007.

[3] J. White, A. Nechypurenko, E. Wuchner, and D. C. Schmidt, "Reducing the Complexity of Optimizing Large-scale Systems by Integrating Constraint Solvers with Graphical Modeling Tools," in *Designing Software-Intensive Systems: Methods and Principles*, P. F. Tiako, Ed., 2007.

[4] openArchitectureWare, "openArchitectureWare," www.openarchitectureware.org, 2007.

[5] Robby, M. Dwyer, and J. Hatcliff, "Bogor: An Extensible and Highly-Modular Model Checking Framework," in *Proceedings of the* 4^{th} *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*. Helsinki, Finland: ACM, September 2003.

[6] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," in *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.

[7] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang, "BPEL4WS Unit Testing: Framework and Implementation," in *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*. Orlando, FL: IEEE Computer Society, 2005, pp. 103–110.

[8] J. H. Hill and A. Gokhale, "Validation of Functional (In)Correctness for Large-scale Component-based Systems using Model-driven Engineering," in *Proceeding of ACM/IEEE* 10^{th} *International Conference on Model Driven Engineering Languages and Systems (MoDELS) (poster session)*, Nashville, TN, September 2007.

[9] ——, "Continuous QoS Provisioning of Large-scale Component-based Systems using Model Driven Engineering," in *Proceeding of ACM/IEEE* 9^{th} *International Conference on Model Driven Engineering Languages and Systems (MoDELS) (poster session)*, Genova, Italy, October 2006.

[10] J. H. Hill, J. Slaby, S. Baker, and D. C. Schmidt, "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: IEEE, Aug. 2006.

[11] J. H. Hill, D. C. Schmidt, and J. Slaby, "System Execution Modeling Tools for Rapid Evaluation of Enterprise Distributed Real-time and Embedded System Quality of Service," in *Designing Software-Intensive Systems: Methods and Principles*, P. F. Tiako, Ed., 2007.

[12] J. H. Hill, S. Tambe, and A. Gokhale, "Model-driven Engineering for Development-time QoS Validation of Component-based Software Systems," in *Proceedings of 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, Tucson, AZ, Mar 2007.

[13] J. Gray, J. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle, "Domain-Specific Modeling," in *CRC Handbook on Dynamic System Modeling, (Paul Fishwick, ed.)*. CRC Press, May 2007.

[14] J. H. Hill and A. Gokhale, "Using Generative Programming to Enhance Reuse in Visitor Pattern-based DSML Model Interpreters," Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, Tech. Rep. ISIS-07-810, June 2007.

[15] K. Chen, J. Sztipanovits, and S. Neema, "Toward a Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA: ACM Press, 2005, pp. 35–43.

[16] K. Chen, J. Sztipanovits, S. Abdelwahed, and E. K. Jackson, "Semantic anchoring with model transformations." in *ECMDA-FA*, 2005, pp. 115–129.

[17] A. Narayanan and G. Karsai, "Using Semantic Anchoring to Verify Behavior Preservation in Graph Transformations," *Electronic Communications of the EASST*, vol. 4, no. 2006, January 2006. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/279.html

[18] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata," *CWI-Quarterly*, vol. 2, no. 3, pp. 219–246, September 1989.

[19] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994. [Online]. Available: citeseer.ist.psu.edu/alur94theory.html

[20] T. Taibi, L. B. Ping, N. S. Wen, L. K. Sing, and C. K. Lim, "Developing a Distributed Stock Exchange Application using CORBA," in *Proceeding of the Student Conference on Research and Development (SCOReD)*, Putraiaya, Malaysia, 2003.

[21] *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 ed., Object Management Group, May 2003.

[22] Microsoft Corporation, "Microsoft .NET Development," msdn.microsoft.com/net/, 2002.

[23] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, pp. 44–51, November 2001.

[24] J. White and D. C. Schmidt, "Simplifying the Development of Product-line Customization Tools via Model Driven Development," in *Proceedings of the MODELS 2005 workshop on MDD for Software Product-lines*, Half Moon Bay, Jamaica, Oct. 2005.

[25] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987. [Online]. Available: citeseer.ist.psu.edu/article/harel87statecharts.html

[26] J. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice Hall, 1981.

[27] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose, *Eclipse Modeling Framework*. Reading, MA: Addison-Wesley, 2003.

[28] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. New York: John Wiley & Sons, 2004.

[29] B. Trask and A. Roman, "Model Driven Engineering Basics using Eclipse," in *Proceeding of ACM/IEEE 9^{th} International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Genova, Italy, October 2006.

[30] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata, Synthesis Lectures in Computer Science*. San Rafael, CA: Morgan and Claypool Publishers, Apr. 2006.

[31] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems," in *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*. San Francisco, CA: IEEE, Mar. 2005, pp. 190–199.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[33] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. B. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *Software Engineering*, vol. 16, no. 4, pp. 403–414, 1990. [Online]. Available: citeseer.ist.psu.edu/harel90statemate.html

[34] B. P. Douglass, "UML Statecharts," www-md.e-technik.uni-rostock.de/ma/gol/ilogix/umlsct.pdf.

[35] D. Huang and H. Sarjoughian, "Software and Simulation Modeling for Real-Time Software-Intensive Systems," in *Proceedings of the Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 196–203.

[36] M. Naughton, J. McGrath, and D. Heffernan, "Real-time Software Modelling using Statecharts and Timed Automata Approaches," in *Proceedings of the IEE Irish Signals and Systems Conference*, Dublin, Ireland, June 2006.

[37] K. G. Larsen, M. Mikucionis, and B. Nielsen, "Online testing of real-time systems using uppaal," in *FATES*, 2004, pp. 79–94. [Online]. Available: springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3395&spage=79

[38] I. A. Niaz, "Code Generation From Uml Statecharts." [Online]. Available: citeseer.ist.psu.edu/635920.html

[39] Y. Gurevich, B. Rossman, and W. Schulte, "Semantic Essence of AsmL," *Theoretical Computer Science*, vol. 343, no. 3, pp. 370–412, 2005.

[40] S. J. Mellor, M. J. Balcer, S. Mellor, and M. Balcer, *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley Professional, May 2002.

[41] S. Nordstrom, S. Shetty, D. Yao, S. Ahuja, S. Neema, and T. Bapty, "The Action Language: Refining a Behavioral Modeling Language," in *Proceedings of the 12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005)*. Piscataway, NJ, USA: IEEE, 2005.

[42] D. Box and D. Shukla, "WinFX Workflow: Simplify Development with the Declarative Model of Windows Workflow Foundation," *MSDN Magazine*, vol. 21, pp. 54–62, 2006.

[43] M. Dutoo and F. Lautenbacher, "Java Workflow Tooling (JWT) Creation Review," www.eclipse.org/proposals/jwt/JWT

[44] Object Management Group, "BPMN Information Home," www.bpmn.org, 2005.

James H. Hill is a 4th year Ph.D. student in the Electrical Engineering and Computer Science Dept 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 H. Hill is a member of ACM.

Aniruddha Gokhale is an Assistant Professor of Computer Science and Engineering in the Electrical Engineering and Computer Science Dept at Vanderbilt University, Nashville, TN. His primary research interests are in investigating synergies between model-driven engineering and middleware to address challenges in the deployment and configuration of distributed real-time and embedded systems with focus on assuring quality of service. He received his B.E. in Computer Engineering from University of Pune, India in 1989; M.S in Computer Science from Arizona State University, Tempe, AZ in 1992; and D.Sc in Computer Science from Washington University in St. Louis in 1998. Prior to his current position, he was with Bell Labs in Murray Hill, NJ. Dr. Gokhale is a member of the IEEE and ACM.