Model-driven Engineering for Development-time QoS Validation of Component-based Software Systems*

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

September 27, 2007

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 lacking significant research deals with the validation of quality-of-service (QoS) properties of component-based systems throughout their development lifecycle instead of waiting until system integration time, which is very late and can be 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 empirical benchmarking code for system emulation and continuous QoS evaluation.

keywords: model-driven system engineering,continuous QoS validation, code generation.

1 Introduction

Model-driven engineering (MDE) [27] techniques are increasingly being used to address many of the development and operational lifecycle complexities of large-scale component-based systems. Advances in MDE techniques for component-based systems to date have focused primarily on (a) structural issues of system development, such as component assembly, packaging, configuration and deployment (e.g., the CoSMIC tool chain [7]), and (b) functional and behavioral issues, such as model checking for functional correctness (e.g., Bogor [26] and Cadena [12]) or runtime validation of performance (i.e., running simulations at design time or 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" [27] 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)). System QoS validation, therefore, can proceed only when all the system components are available and deployed in the runtime infrastructure. Moreover, waiting too late in the development lifecycle 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.

*This work was supported by Raytheon IRAD

A promising solution to address the challenge of evaluating QoS at all stages of development entails accurately emulating system components for QoS validation while the "real" components are still being developed. This paper describes our novel MDE-based solution to address the challenges of serialized phasing and QoS validation across the development lifecycle. First, we demonstrate how the problem of serialized phasing can be overcome by emulating component "business logic" using domain-specific modeling languages (DSMLs) [8]. The behavior in our DSML is captured using the formalisms of I/O automata [18] and can be parametrized with executable operations (i.e., workload). Generative programming tools [4] associated with the DSMLs can synthesize empirical benchmarking code for system emulation and QoS evaluation. The QoS evaluation is carried out using the QoS benchmarking framework for component-based systems called the Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS) [14].

Paper Organization. The remainder of this paper is organized as follows: Section 2 introduces a case study we use to describe the challenges in realizing a solution for continuous QoS evaluation; Section 3 describes the structure and functionality of our DSMLs for emulating component behavior and workload; Section 4 explains how we integrated our DSMLs with an existing structural DSML to facilitate code generation for QoS evaluation; Section 5 compares our work with related research; and Section 6 presents concluding remarks.

2 Challenges in Overcoming the Serialized-phasing Barrier

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

2.1 A Distributed Stockbroker Application Case Study

We use a representative example drawn from the financial domain [28] as a case study to illustrate the serialized phasing problem and how our research artifacts described in this paper enable us to provide continuous QoS validation [13]. 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*.

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 1 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 times. Evaluating system design decisions on the target architecture to locate deployments (i.e., placement of components on hosts) and configurations (i.e., setting of compo-

nent attributes) that meet the expected QoS therefore has to wait until all the "real" components are available.

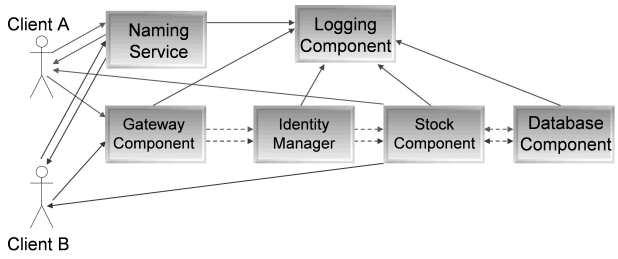

Figure 1: High-level structural composition of the Distributed Stock Application.

Usage Patterns by User Type

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

Table 1: Predicted usage pattern of the Distributed Stock Application based on user type.

The application components of DSA are implemented as Lightweight CORBA Component Model (CCM) [24] 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`.

2.2 Impediments to Overcoming the Serialized-phasing Barrier

To achieve the vision of continuous 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: Emulating business logic** – The emulated components must resemble their the counterparts in both supported interfaces and behavior. Moreover, the emulation environment should allow seamless replacement of faux components with real components as they become available. 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 completed they should replace the emulated components to achieve more accurate QoS metrics.
- **Challenge 2: Realistic mapping of emulated 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.
- **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 [20], or use multiple modeling tools [17, 30], then we should be able to reuse the same concepts and models.

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

3 Domain-specific Modeling Languages for Continuous 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 two domain-specific modeling languages (DSMLs) [8] called the Component Behavior Modeling Language (CBML) and Workload Modeling Language (CBML). CBML is a DSML for capturing the behavior of a component and WML is a DSML for parameterizing the behavior with "realistic" application-level operations. The remainder of this section discusses both languages in detail explaining how these help resolve Challenges 1 and 2 discussed in Section 2.2 in the context of the case study described in Section 2.1.

3.1 The Component Behavior 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 2.2), 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 [1], StateCharts [10] and PetriNets [25]), 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 behavior 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 Behavior 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) [17], 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., Generic Eclipse Modeling System (GEMS) [30]). 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 and simulation tools. Our current efforts focus primarily on generating source files for emulation tools (see Section 4).

3.1.1 Structure of CBML

As explained in Section 3.1, 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 (illustrated in Figure 2) that will preserve its formal semantics. Users of CBML do not need prior knowledge of I/O automata in order to use CBML.

Figure 2: Primary elements for constructing behavior models in CBML.

Figure 3 shows the realization of the CBML artifacts illustrated in Figure 2 using the DSA database component as an example. In CBML, all behavior 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 behavior specification is defined by a sequence of *Action* to *State* transitions (similar to I/O automata). For example, the behavior model for the database component in Figure 3 illustrates that an input action causes a query for stock information.

To specify the end of a behavior sequence, a *Finish* connection (i.e., the dashed line) is used to connect the final *State* to the starting *Input Action*. We

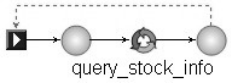

Figure 3: Example CBML behavior model in GME.

require this connection because we allow sharing of behavior 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. The explicit finish connections therefore help resolve ambiguity when determining where each user type's behavior terminates.

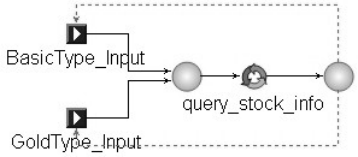

Figure 4: Example of sharing behavior in CBML.

Specifying output actions in CBML

I/O automata defines behavior as an input action that causes a series of "internal" operations and results in a set of output actions, if any. To be consistent with I/O automata semantics, we allow behavior specifications to include output operations, however, output actions have the same *modeling* semantics as I/O automata internal operations (e.g., CBML *Action* element). We made this design choice because, similar to CBML action elements, output actions can also have a series of action-to-state transitions after completing a single output action.

Figure 5 illustrates an example behavior model with output actions, which are represented by the two rightmost squares with the triangle, for the database component in the DSA. After the component completes its query to the database for stock information, it sends the information back to the requester, and sends a status message to the logging component.

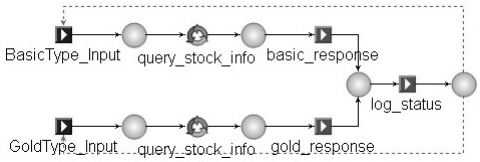

Figure 5: Example CBML behavior model with output actions.

Preconditions, postconditions, and variables in CBML

CBML allows a user to define variables in behavior models to stay consistent with the I/O automata semantics. The purpose of a variable is to preserve information that represents the current state of the system or component. As illustrated in Figure 6, a variable is represented by the element with the star image. Users use variables in their behavior 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. This allows developers to create more "realistic" behavior 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.

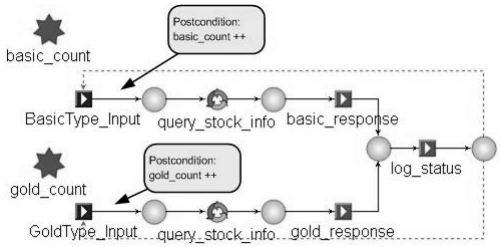

Figure 6: Example CBML behavior model with variables.

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 7:

- **Environment Events** – 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** – 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.

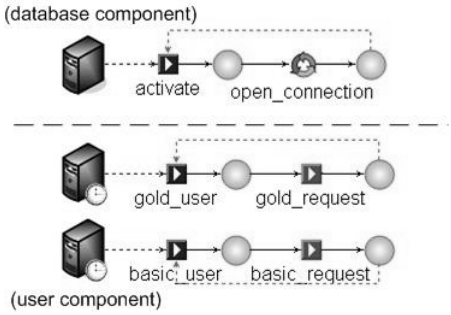

Figure 7: CBML's domain-specific extensions to I/O automata

In the context of the DSA, when the database component is activated it creates an initial connection to the database (illustrated in Figure 7). 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 7).

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 behavior 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 [10]. To address this issue we defined the following usability extensions, which do not affect the underlying I/O automata semantics:

- **Composite Action** – is a modeling element that contains other actions. It allows developers to create groups of action-to-state sequences that can help reduce the amount of clutter in the model. A composite action has the same modeling semantics as a regular action, however, we defined a constraint that requires composite actions to contain only a single input action. 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.
- **Repetitions** – is an attribute of an *Action* element that specifies how many times to repeat the operation. This prevents the same action from having to be specified multiple times in order to achieve repetition. It is clear that setting this value to zero implies the action is disabled. This allows developers to seamlessly bypass an action temporarily without actually removing it.

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].

3.2 The Workload Modeling Language

The Component Behavior Modeling Language (CBML) described in Section 3.1 gives developers the ability to model behavior via generic actions and properties. For analysis techniques, such as simulation, CBML is enough to capture the behavior of the component (e.g., its actions, states, and respective transitions), which can be interpreted to define configuration files for simulation tools. For emulation purposes, however, these actions do not exemplify the "business logic" of components because it does not capture the workload of reusable objects within a component (e.g., objects and their methods). Moreover, when defining the workload of components using CBML, it is "hard" to specify realistic workloads that map to executable operations for an emulated component. To address this challenge (i.e., Challenge 2 in Section 2.2) we developed the Workload Modeling Language (WML).

WML is a middleware and hardware platform-independent and programming language-independent DSML that allows developers to define workload generators that contains actions to represent realistic operations (e.g., memory allocations/deallocations and database transactions) at a high-level of abstraction. Model interpreters associated with WML parse the constructed models and use generative programming techniques to map the abstract representation to executable operations in the target programming language and platform (see Section 4).

We implemented WML in GME, but similar to CBML, it can be ported to any MDE tool that supports metamodel specification. The remainder of this section discusses WML in detail.

3.2.1 Structure of WML

WML is a DSML that allows developers to create workload generators (called "workers") with executable actions for emulation. Figure 8 illustrates the compositional overview of WML. We designed WML using a hierarchical structure that resembles common object-oriented programming packaging techniques to be consistent with conventional component technologies.

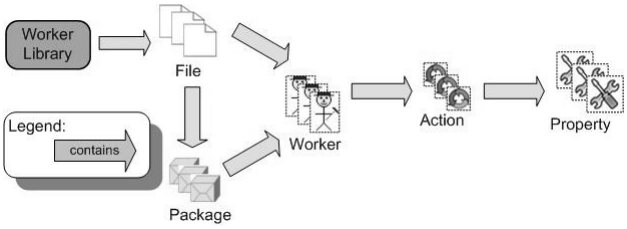

Figure 8: High-level compositional overview of WML.

The outermost containment elements in WML are *library* elements. Library elements represent reusable containers (e.g., modules) for grouping common *workers*. The library elements are composed of multiple *files*, which represent a concrete location on disk that defines its contained workers (recall that workers are workload generators). File elements can contain *packaging* elements that act as a scoping mechanism so that *workers* can have the same name and appear in the same file (similar to C++ namespaces or Java packages). *Workers* contain executable actions that represent its "business logic" operations (or workloads). Lastly, actions can contain optional *properties* that define configurable parameters (e.g., input arguments) for the action executed by the parent.

3.2.2 Parameterizing CBML with executable operations

When WML is integrated with CBML, it enables developers to model the component behavior using executable operations. From a modeling perspective, *workers* in WML have the same modeling semantics as *variables* in CBML, and worker *actions* in WML have the same modeling semantics as *actions* in CBML. This design feature allows us to integrate WML with CBML.

Figure 9 illustrates the behavior model of the database component from Figure 6 in Section 3.1.1 that has been parameterized with WML actions. The top portion of the image illustrates the WML composition for a database worker. In the bottom portion

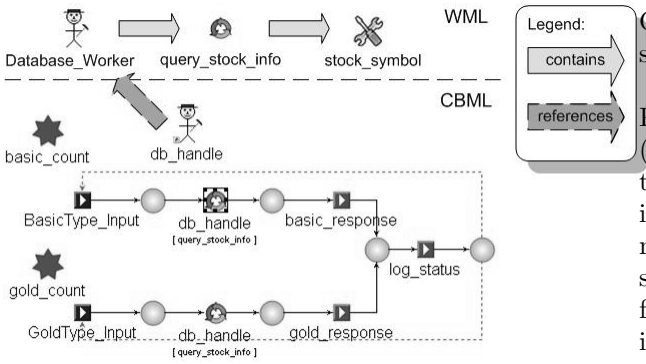

Figure 9: Example CBML model parameterized with WML actions.

of the image, the actor (i.e., `db_handle`) is a *variable* that references the database worker. The *action* is a modeling instance of the worker action in the top portion of the image whose name must match the name of its parent worker variable. We made this design requirement because it (1) helps resolve ambiguity when determining what action belongs to what parent since it is possible to include the same worker variable type multiple times in a behavior model, and (2) reduces modeling clutter as opposed to explicitly creating a directed connection between a parent and its action.

4 Technology Independent Approach to Continuous QoS Validation

In Section 3 we described two behavioral DSMLs: CBML and WML that illustrated how integrating both languages allowed us to emulate the behavior (Challenge 1 of Section 2.2) and mapping the behavior to realistic operations (Challenge 2 of Section 2.2). Although WML allowed us to parameterize the generic actions in CBML with executable operations, these models are insufficient to generate emulation code directly without knowing the structural composition of the system and its components since the latter determines the end-to-end workflows whose QoS validation is more interesting and important to system developers.

We therefore integrated CBML and WML with the Platform Independent Component Model Language (PICML) [2] because the latter captures the structural aspects of a system and its components, which is necessary when generating source code for components that resemble the real counterparts. Moreover, since both PICML and CBML/WML 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 2.2).

Although we chose PICML as the structural DSML to integrate CBML and WML, the concepts presented in Section 4.1 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 and WML with existing languages (e.g., PICML in CoSMIC), and how our approach to generating emulated logic for components that mimics their real capabilities is decoupled from the underlying platform and programming language technology.

4.1 Integrating Behavioral and Structural DSMLs

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. 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 defined a set of "connector elements" that enable developers to connect the I/O elements in the structural model with their corresponding I/O elements in the behavioral model.

Figure 10 illustrates how structural DSMLs (e.g., PICML) which define components that have I/O ports and behavioral DSMLs (e.g., CBML and WML)

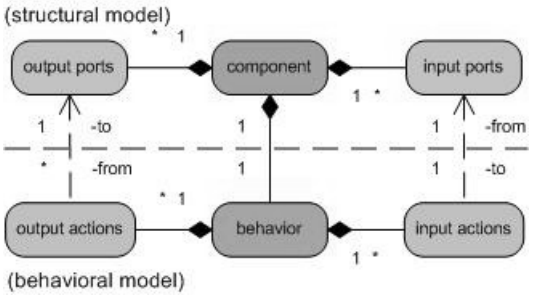

Figure 10: Conceptual model of integrating behavioral and structural DSMLs.

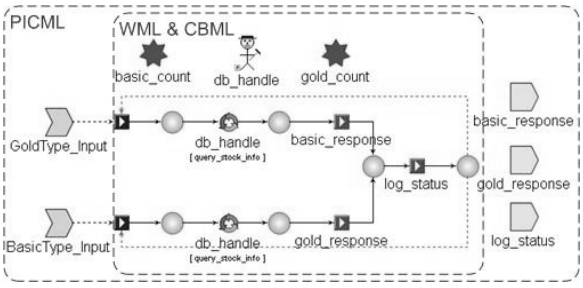

Figure 11: Realization of integrating CBML and WML with PICML in CoSMIC.

that have I/O actions can be integrated by having the structural DSML "contain" the behavioral DSML. In particular, we require a component to contain the behavior. Additionally, we define a modeling connection between the input port and input action, but require that the name of the output action match the name of the corresponding output port. 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 11 shows how the integration of the DSMLs is realized. The outer rectangle of Figure 11 illustrates the PICML model of the database component. The inner rectangle highlights the same database component with CBML and WML from Figure 9 integrated into PICML, thus allowing us to model the same behavior exemplified in Section 3 with its respective structure (e.g., interface and attributes).

4.2 Code Generation for Emulation

This section describes our approach for achieving code generation for emulation, which enables us to conduct continuous QoS validation during system development. Our current effort allows developers to generate emulation code for the *Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS)*, however, our code generation architecture is not dependent on CUTS (e.g., Challenge 3 in Section 2.2). Figure 12 illustrates a conceptual model of our code generation architecture, which is composed from three technology independent, but language dependent layers of abstraction:

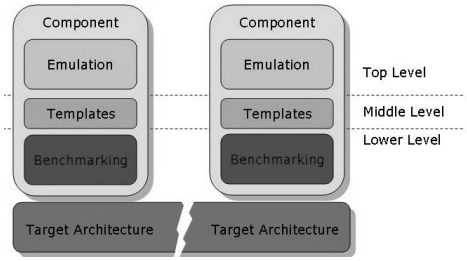

Figure 12: Code generation architecture for emulation.

- **Emulation** - This layer represents the application layer's "business logic". The elements in WML used to parameterize the CBML behavior are mapped to this layer when model interpreters parse the model. For example, the *query_stock_info* action is generated at this layer in C++ code.
- **Templates** - This layer acts as a bridge [6] between the upper emulation layer and lower benchmarking layer. This allows both layers to evolve independently of each other. For example, if we want to provide support for other benchmarking frameworks we do not have to alter the generated code because the templates will provide the mapping. Likewise, if we ported the DSA to a different technology (or language) the code generator can tailor the source code to plug

into this layer given we support the target programming language.

- **Benchmarking** - This layer represents the underlying benchmarking framework (e.g., CUTS). Methods in this layer are invoked by the template layer above to capture workload metrics, such as execution timing of a database query by the database component, or response time of each user type in the context of the DSA.

Lastly, the encapsulating object for each of the three layers is the actual *component* hosted by the *target architecture*, which is language and technology dependent. The component is generated so that is has the same structure as its "real" counterpart (e.g., same interfaces and attributes). Figure 13 illustrates the generated code for a portion of the database component in the DSA. As illustrated in Figure 13, the *push_BasicType_Input* method is the realization of implementing the *BasicType_Input* input event port in CCM. Each line of source code represents the WML actions used to parameterize the CBML behavior. The *record* is the template object that allows the emulation operations to be adapted for monitoring and analysis by the CUTS benchmarking framework.

```
void DatabaseComponent::push_BasicType_Input (
  QueryEvent * ev ACE_ENV_ARG_DECL_WITH_DEFAULTS)
  ACE_THROW_SPEC ((::CORBA::SystemException))
{
  // get activation record for this thread
  CUTS_Activation_Record * record = CUTS_THR_ACTIVATION_RECORD ();
  this->basic_count_ ++;

  record->perform_action_no_logging (
    CUTS_Database_Worker::query_stock_info (this->db_handle__));

  CUTS_CCM_Event_T <OBV_QueryResponse> __event_100000028__;
  this->context_->push_basic_response (__event_100000028__.in ());

  CUTS_CCM_Event_T <OBV_LogStatus> __event_100000029__;
  this->context_->push_log_status (__event_100000029__.in ());
}
```

Figure 13: Excerpt of generated code from a PICML model extended with CBML and WML.

Generation for Simulation. Although our efforts currently focus on generating emulation code, it is possible to create model interpreters that generate configuration files for simulation tools such as UPPAAL [16]. This will allow developers to utilize the features of a simulation tool such as validating correctness, evaluating preconditions and postconditions, and checking for reachability [15]. Moreover, it will alleviate the time and effort required to manually produce these configuration files, which can be error prone and tedious.

5 Related Work

Statecharts [10] gained widespread usage when they were integrated with the STATEMATE [11] modeling tool, and since then a variant has became part of UML (i.e., UML Statecharts) [5]. Similar to CBML, statecharts can be used to describe behavior of large complex systems. CBML extends Statecharts by clearly separating component behavior from workloads using WML. The generative techniques associated with variants of statecharts are targeted towards simulation and runtime verification [15, 21]. Our generative techniques can be extended to simulation and runtime verification tools [16] as well, however, it extends UML statecharts efforts [22] because it facilitates seamless replacement of the emulated components (i.e., components generated from behavior models) with the real components as they become available. Furthermore, our generative techniques and concepts are not tied to a specific technology or tool, whereas the technique presented in [22] et al., is bound to a specific tool.

The Abstract State Machine Language (AsmL) [9] 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 and WML. Furthermore, users of CBML and WML 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).

Executable UML (xUML) [19] and the Action Lan-

guage [23] are both for defining workload that can map to the desired target architecture. WML is orthogonal to both xUML and the Action Language efforts, however, WML operates at higher level of abstraction. xUML and the Action Language require developers write abstract implementation code, which requires knowledge of programming semantics, whereas WML leverages pre-existing objects and methods (i.e. workload generators) that are not defined by the user for code generation.

WinFX Workflow [3] is a 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.

6 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 two modeling languages, namely CBML and WML, that capture the behavior of application components at a high-level. We then integrated these DSMLs with PICML, which models structural properties of applications. Lastly, we used model interpreters to map the behavior specifications to executable operations that leverage existing emulation frameworks, such as CUTS.

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 emulation. Likewise, as the real application components are ready, they can replace the emulated components and their impact on system QoS can be observed. We expect the results of real versus emulated components to match provided the behavioral models of the emulated components approximate the real component behavior closely.

6.1 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 WML 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 code and configuration files for emulation and simulation.
- Separating the workload, behavior, and structural models allows all to evolve independently of each other. Moreover, it encourages the same behavior model to be supported in multiple structural models to increase portability, flexibility, and usability.
- Using generative programming with templates that are parameterized by actions from behavioral models allows the DSML to easily be adapted to different environments of execution (e.g, benchmarking environments or real-world deployment).

6.2 Future Work

Although our approach of integrating a behavior and workload 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.

- As the "real" components become available and replace the emulated components, it is ideal to capture workload metrics of the real component. We therefore need to extend the current capabilities of both modeling languages and code generators to handle evaluation of real components (i.e., benchmarking them using "realistic" input data).
- The workload generators (i.e., *workers*) in WML resemble class objects in object-oriented programming languages. We therefore need to extend the capabilities of WML such that existing class objects in the target programming language can be used in WML models. Moreover, these extensions will enable "real" implementation code to be generated directly from models.

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] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] 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 *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 190–199, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

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

[4] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

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

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

[7] 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.

[8] 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.

[9] Y. Gurevich, B. Rossman, and W. Schulte. Semantic Essence of AsmL. *Theoretical Computer Science*, 343(3):370–412, 2005.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[11] 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*, 16(4):403–414, 1990.

[12] 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.

[13] J. H. Hill and A. Gokhale. 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.

[14] 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, August 2006.

[15] 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)*, pages 196–203,

Washington, DC, USA, 2004. IEEE Computer Society.

[16] K. G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using uppaal. In *FATES*, pages 79–94, 2004.

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

[18] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

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

[20] Microsoft Corporation. Microsoft .NET Development. `msdn.microsoft.com/net/`, 2002.

[21] 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.

[22] I. A. Niaz. Code Generation From Uml Statecharts.

[23] 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, 2005. IEEE.

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

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

[26] 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, September 2003. ACM.

[27] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[28] 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.

[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] 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.