Towards A QoS Modeling and Modularization Framework for Component-based Systems

Sumant Tambe, Akshay Dabholkar, Aniruddha Gokhale, Amogh Kavimandan
Department of EECS, Vanderbilt University, Nashville, TN, USA
{sutambe, aky, gokhale, amoghk}@dre.vanderbilt.edu

Abstract

Current domain-specific modeling (DSM) frameworks for designing component-based systems provide modeling support for system's structural as well as non-functional or quality of service (QoS) concerns. However, the focus of such frameworks on system's non-functional concerns is an after-thought and their support is at best adhoc. Further, such frameworks lack strong decoupling between the modeling of the system's structural composition and their QoS requirements. This lack of QoS modularization limits (1) reusability of such frameworks, (2) ease of their maintenance when new non-functional characteristics are added, and (3) independent evolution of the modeling frameworks along both the structural and non-functional dimensions.

This paper describes Component QoS Modeling Language (CQML), which is a reusable, extensible, and platform-independent QoS modeling language that provides strong separation between the structural and non-functional dimensions. CQML supports independent evolution of structural metamodel of composition modeling languages as well as QoS metamodel. To evaluate, we superimpose CQML on a purely structural modeling language and automatically generate, configure, and deploy component-based fault-monitoring infrastructure using aspect-oriented modeling (AOM) techniques.

1 Introduction

Recent advances in domain-specific modeling (DSM) [14] have resulted in DSM tool suites for designing large, component-based software systems with multiple quality of service (QoS) requirements, such as predictable latencies, fault-tolerance and security. Recent successes with DSM tools in this area include the Embedded Systems Modeling Language (ESML) [9] for avionics mission computing, SysWeaver [5] for embedded systems, and our earlier work on the Platform Independent Component Modeling Language (PICML) [2] for a range of distributed, real-time systems. These DSM tools provide support for component-based software engineering (CBSE) [3] wherein systems can be modeled by composing multiple different components, each encapsulating a reusable unit of functionality.

Despite the number of benefits of these DSM tools and techniques, however, designing operational QoS-intensive systems remains a significantly hard problem due to multiple crosscutting non-functional characteristics (*i.e.*, the secondary concerns) that must be satisfied simultaneously along with system's functional composition (*i.e.*, primary) concern.

As an example of a non-functional requirement that affects system's structural dimension consider fault-tolerance requirements, such as various replication styles (active, passive). Such fault-tolerance requirements may be specified at several different levels of granularity, such as per-component basis, across a group of components, and across nested component groups. Replication is the most widely used pattern for developing highly available systems, inherently requires additional copies of components that must be composed with original business components. Moreover, for detecting failures of distributed components liveness monitoring infrastructure must be composed parallel to the business components that are monitored. This illustrates the scattering of the fault-tolerance concern across the system functional composition dimension.

Scattering (crosscutting) of fault-tolerance concerns is not only observed along the system's structural concern but also along other non-functional concerns, such as deployment. The impact of the fault-tolerance concerns on the deployment concern is also non trivial since the deployment must now account for the placement of the replicated business components, proxy components as well as the placement of the monitoring components to enable timely detection of failures of business components.

To assist in designing systems where non-functional concerns crosscut with structural concerns, DSM tools must provide strong decoupling between system's structural concerns, deployment concerns, and non-functional concerns and combine them when the final system is realized. Such

decoupling should not only provide different views for different concern models but should also enable evolution of the modeling capabilities of each view independently. Evolution of the modeling capabilities of a concern view often requires enhancements to the meta-model of the view. Supporting independent evolution of meta-models of each concern view shortens the development lifecycle by allowing parallel enhancements to the modeling capabilities (*i.e.*, the meta-model) and models pertaining to the view.

Moreover, platform-independent notion of QoS requirements is largely independent of the structural capabilities of the chosen implementation platform. For example, the CORBA Component Model [12] (CCM) is considered to be a richer component model compared to Enterprise Java Beans (EJB) as the former borrows several concepts from the latter and adds its own, such as a wider variety of component types, a notion of assembly, required and provided interfaces using ports, and event-based communication. This structural variability is clearly visible in platform specific modeling tools. However, such variability in the structural capabilities of the contemporary component platforms need not prevent their corresponding DSM tools from having a platform-independent modeling support for QoS such as fault-tolerance, timeliness, authentication and authorization and network level QoS. All of which have little or no bearing on the structural capabilities of the platform. However, contemporary DSM tools are based on ad-hoc designs of meta-models for modeling QoS, which couple them tightly with structural capabilities, prevent their reuse in other component platforms and limits extensibility. Moreover, QoS and structural modeling capabilities (meta-models) are hard to evolve independently of each other.

This paper describes our solution to address these limitations of DSM design tools for CBSE. We describe Component QoS Modeling Language (CQML), which is a reusable and platform-independent framework developed using the Generic Modeling Environment (GME) [10]. CQML is designed to superimpose on a wide range of system structural composition modeling languages as long as they conform to a small set of invariant properties defined by CQML.

CQML has an extensible QoS modeling framework that allows declarative QoS requirements to be associated with structural component models. In this paper we demonstrate how QoS modeling capabilities can be superimposed on a purely structural modeling language. Moreover, we show how CQML's fault-tolerance modeling capabilities can be used to automatically generate runtime fault monitoring infrastructure using the structural modeling capabilities of the underlying language.

The remainder of this paper is organized as follows: Section 2 describes extensible QoS modeling capabilities of CQML; Section 3 evaluates the capabilities of CQML; Section 4 describes related research; and Section 5 concludes the paper.

2 Solution: Extensible QoS Modeling Using CQML

In this section we describe the design of the Component QoS Modeling Language (CQML), which is a platform-independent, QoS modeling framework that allows component-based system developers and designers to express QoS design intent at different levels of granularity using intuitive visual representations. CQML has been developed using the Generic Modeling Environment (GME) [10] toolkit. CQML is capable of separating system's QoS concerns from the primary concern: structural composition. It also supports QoS modeling for a multitude of component-based platforms because CQML depends only on the commonalities present across them. Figure 1 shows the process of using CQML. We now describe how CQML uses this process to resolve the challenges described in the previous section.

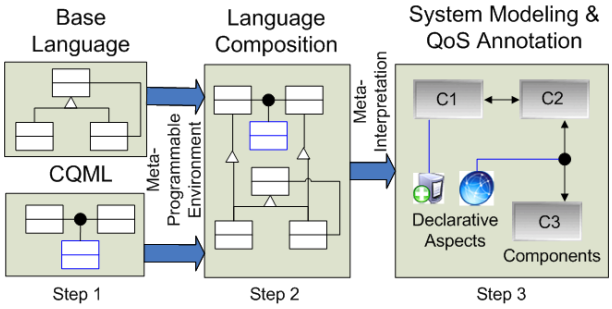

Figure 1: Process Model for Reusing CQML for QoS Modularization and Weaving

2.1 Identifying Invariant Properties of Component-based Structural Modeling Languages

Our focus is on general component-based systems, which are composed using multiple components orchestrated to form application workflows. Contemporary component models often have first class support for primitives, such as components, connectors, and methods. The structural artifacts of a component-based system can be realized using these primitives in a language specifically designed for modeling system structure.

Since CQML is aimed specifically at modularizing non-functional concerns of component-based systems in a platform-independent manner, CQML requires an underlying base composition modeling language that allows construction and manipulation of platform-specific structural models. Many platform-specific as well as platform-independent component structural modeling languages, such as Embedded Systems Modeling Language(ESML) [9] for embedded systems, J2EEML [17]

for Enterprise Java Beans, and Platform Independent Component Modeling Language (PICML) [2] for Light-weight CORBA Component Model [12] (LwCCM) exist today that capture various composition semantics. In this paper we have focused on languages developed using GME since CQML is also developed using GME. However, the concepts behind CQML can be applied in other tool environments.

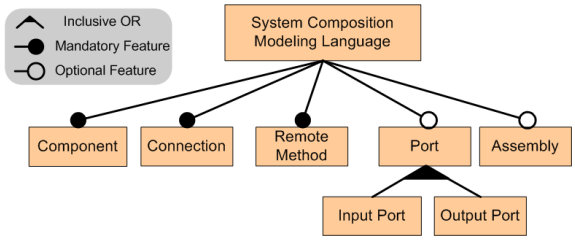

Figure 2: A Feature Model of Composition Modeling Language

We refer to such a structural modeling language as *system composition modeling language* or *base language* in short. We formalize the features of a *base language* in a feature model [4] shown in Figure 2. CQML is designed taking into account the mandatory and optional features present in such languages. The base language should have first class modeling support for components, connectors, and remotely invocable methods at the minimum. ESML, J2EEML, and PICML support all the mandatory entities mentioned in Figure 2 and therefore these languages can play the role of a base language for CQML as shown in step (1) in Figure 1. In step (2), meta-modeling composition techniques are used to "mix-in" the metamodel of CQML with that of the base composition modeling language without affecting the syntax and semantics of the structural modeling language. In step (3), the composed meta-model is used to create the instances of the composition modeling language, which has enhanced QoS modeling capabilities due to CQML.

2.2 Extensible Design of CQML

Based on the feature model of component-based modeling languages, CQML builds an extensible QoS modeling layer. CQML has an ability to associate declarative QoS characteristics to one or more of the invariant properties of the underlying base language. We have designed several declarative QoS characteristics that are applicable to a general class of component-based systems. We have developed (1) *FailOverUnit* [16], which modularizes fault-tolerance requirements of one or more components and assemblies, (2) *PortSecurityQoS*, which modularizes security aspects of port based communication, (3) *NetworkQoS* [1], which modularizes network level QoS requirements while invoking remote methods. Some examples of the above concrete QoS characteristics are shown in Figure 3. A *FailOverUnit* is used to annotate component A as a fault-tolerant component. For connections between component B and C, network level QoS attributes (*e.g.,* priority of communication traffic) are associated using *NetworkQoS* modeling element. In this paper, we do not explain the semantics of all the above concrete QoS characteristics in detail, however, interested readers are encouraged to read [1] and [16] to read more on *NetworkQoS* and *FailOverUnit* modeling respectively.

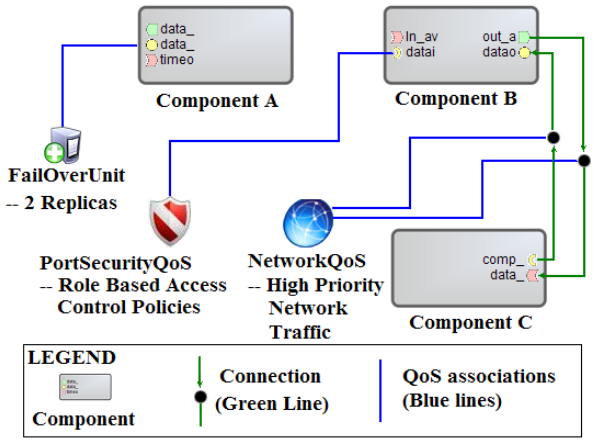

Figure 3: Declarative QoS Modeling Capability of CQML

To support evolution of QoS metamodel without affecting the structural metamodel, CQML defines a set of abstract QoS elements: *Component-QoS*, *Connection-QoS*, *Port-QoS*, *Assembly-QoS* and *Method-QoS*. As the name suggests, each abstract QoS element is associated with its corresponding element from the base language. For example, abstract *Component-QoS* can be associated with a *Component* in the base language. Moreover, all the concrete QoS models (*e.g., FailOverUnit*) must be derived from one or more abstract QoS types as shown in Figure 4.

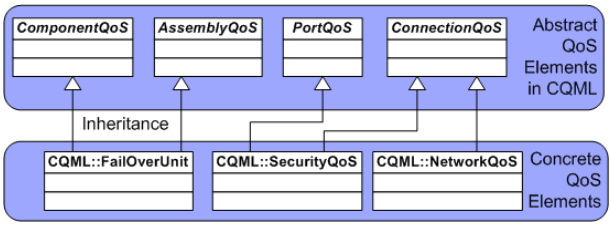

Figure 4: Simplified Meta Model of CQML

CQML can be extended with new concrete declarative QoS modeling capabilities by inheriting from the basic set of abstract QoS elements. To enhance CQML with a concrete QoS characteristic, a language designer has to ex-

tend the metamodel of CQML at the well-defined points of extension represented by the five abstract QoS elements. The concrete QoS elements simply derive from the abstract QoS elements defined in CQML. By doing so the concrete modeling entities inherit the abstract syntax, static semantics, relationships, integrity constraints, and visualization constrains of the abstract QoS entities defined in the meta-model of CQML. For example, as shown in Figure 4, *FailOverUnit* inherits association constraints from the abstract *ComponentQoS* and *AssemblyQoS*. Therefore, *FailOverUnit* can be associated with components and assemblies only and never with ports or connections.

View separation of QoS elements and structural elements is achieved by controlling the visibility of concrete QoS modeling elements using visualization constraints defined on abstract QoS elements. CQML defines visibility constraints on them such that they project QoS concerns in the *QoS* view of GME model editor, which is different from the view where structural concerns are edited and manipulated. All the concrete QoS elements inherit association and visualization constraints from one or more abstract QoS elements defined by CQML as shown in Figure 4.

Along with the basic five abstract QoS elements, these visualization and association constraints constitute the generic QoS modeling framework of CQML. Although designing a new language construct – in this case a new QoS characteristic – is an extremely thoughtful process, a significant portion of design decisions are already taken for the language designer in the generic QoS modeling framework of CQML. The reuse promoted by CQML design and its generic QoS entities thus lends itself to easier component-based systems modeling enhancements. It prevents reinvention of previously designed artifacts for every new QoS concern that is added.

3 Evaluation

Rationale. An important responsibility of a fault-tolerant system is to monitor the running system for faults and when faults are detected they must be reported to higher level components so that appropriate recovery procedure can be initiated. Developers of fault-tolerant DRE systems must also reason about how the monitoring subsystem will be deployed and configured so that failure of business components can be detected and reported in a timely and reliable way. This additional responsibility of designing, deploying, and configuring monitoring subsystem delays developers' main task of developing business logic. Therefore, an automated support for generating, deploying, configuring liveness monitoring infrastructure is highly desirable.

Methodology. To demonstrate the capabilities of CQML, in this section we present an AOM solution to automatically generate, deploy, and configure liveness monitoring infrastructure for fault-tolerant component-based systems from their requirements. We use CQML's *FailOverUnit* modeling capability to capture fault-tolerance requirements of one or more components. In this paper, we use it as a way of annotating a group of components for which runtime monitoring infrastructure should be generated.

Our solution uses Constraint Specification Aspect Weaver (C-SAW) [15], which is a generalized model-to-model transformation engine for manipulating domain-specific models, which is implemented as a plug-in for the Generic Modeling Environment. It can also be used to instrument structural changes within the model according to some higher-level requirement that represents a crosscutting concern. We used PICML as our base structural modeling language and we composed CQML's QoS meta-model with the structural meta-model of PICML.

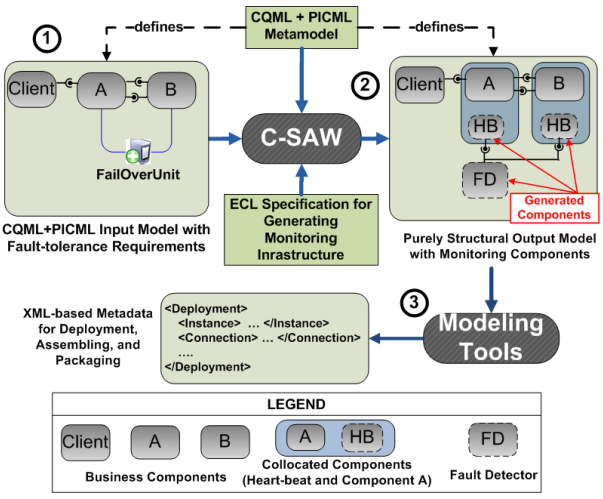

Figure 5: Automatic Weaving of Monitoring Components Using Embedded Constraint Language Specification

As shown in Figure 5, we have developed model transformation specifications for CQML models using C-SAW's input language: Embedded Constraint Language (ECL). These specifications transform CQML models (shown by (1) in Figure 5) with *FailOverUnit* into structural models (shown by (2) in Figure 5) containing monitoring components, their interconnections, and their deployment information. The model-to-model transformation bridges the gap between the higher-level fault-tolerance requirements captured using *FailOverUnit* and lower-level structural models used by existing modeling tools, such as deployment, configuration and packaging tools that generate platform-specific metadata based on structural models. Such a transformation requires several steps, including (1) generating models of monitoring components, (2) generating the necessary interconnections between the instances of monitoring components, and (3) generate deployment and configuration models for instances of monitoring components so

that they are deployed along with the business components when the system is deployed.

Algorithm 1 Transformation Algorithm for Generating Monitoring Infrastructure

1: M : Systems's structural model with annotations.
2: D : Deployment model of the system
3: M_e : Extended M with monitoring components
4: D_e : Deployment model of M_e
5: c : A business component
6: S_c : A set of collocated components such that $c \in S_c$
7: HB_c : Heartbeat component monitoring c
8: F : Fault Detector component.
9: **Input:** M, D
10: **Output:** M_e, D_e (Initially empty)
11: **begin**
12: $M_e := M$
13: $D_e := D$
14: $S_F := \emptyset$
15: $F :=$ New fault detector component
16: $M_e := M_e \cup F$
17: $S_F := S_F \cup F$
18: $D_e := D_e \cup S_F$
19: **for** each component c in M **do**
20: **if** a FailOverUnit is associated with c
21: **let** $HB_c :=$ New heartbeat component for c.
22: $M_e := M_e \cup HB_c$
23: **let** $i :=$ New connection from F to HB_c.
24: $M_e := M_e \cup i$
25: **let** $c \in S_c$ and $S_c \in D$
26: $S_c := S_c \cup HB_c$
27: $D_e := D_e \cup S_c$
28: **endif**
29: **end for**
30: **end**

The algorithm behind the transformation is shown in Algorithm 1. The transformation accepts system's structural model and a deployment model as input and produces an extended structural model with monitoring components and an extended deployment model with placement of monitoring components as output. A *deployment model* can be viewed as a simple mapping of components to physical hosts in a system. Components are called *collocated* components when they are hosted in the same process on the same host. When a process or a host dies, all the components hosted in that process/host become unavailable, which can be detected remotely using monitoring infrastructure.

The algorithm begins with a copy of system's structural and deployment model in the corresponding extended models. For every structural model, a new Fault Detector component is added in the extended model along with its placement in the deployment model. Followed by that, for every business component in the original structural model, a new *Heartbeat* component is added that is collocated with the business component. The collocated components are placed in the same process as that of the business component at run-time. A connection from the *FaultDetector* component to every new *Heartbeat* component is created so that the former can poll the liveness of the later at runtime. As a result of the algorithm, monitoring components are weaved-in the original structural and deployment models of the system.

The above algorithm that we developed for generating monitoring components for PICML models can be used unchanged to generate monitoring components for J2EEML as well as ESML models. In that sense, strong separation of structural modeling from QoS modeling allows us to write generic model transformation algorithms using ECL that work across variety of structural modeling languages as long as they satisfy the minimal invariant properties specified in Section 2.1.

The actual implementation of the *Heartbeat* and *FaultDetector* components is provided as a library using Component Integrated ACE ORB (CIAO), which is our open-source implementation of OMG's Light-weight CORBA Component Model specification. The component library uses the OMG's IDL 3.0 interfaces shown in Listing 1. The *FaultDetector* component periodically invokes *isAlive* remote method on all the instances of the *Heartbeat* component. The implementation of *isAlive* method in *Heartbeat* component returns true indicating the fact that it is "alive". If a *Heartbeat* component does not respond within a configurable timeout period, the *FaultDetector* concludes that the *Heartbeat* component and the collocated business component have failed. It initiates a recovery procedure after detecting the failure.

Listing 1 OMG's IDL 3.0 Interfaces Used by The Fault Monitoring Infrastructure Generated by ECL Transformation

```
module Monitor {  // A module defines a namespace.
interface Monitorable { // An interface for checking liveness.
   bool isAlive(void);  // Returns true if component is alive.
};
component Heartbeat { // Implements Monitorable interface.
   provides Monitorable alive; // Used by the FaultDetector.
};
component FaultDetector { // Polls liveness
   requires Monitorable poll; // Uses Heartbeat components.
};
}
```

Finally, existing model interpreters for generating valid XML-based metadata (shown by Step 3 in Figure 5) are invoked on the weaved-in structural and deployment model to generate packaging and deployment descriptors.

4 Related Work

Capturing QoS specifications at design-time has long been a goal of researchers [7, 18]. A prior effort, called Component Quality Modeling Language [18], developed by Aagadel is a platform-independent, general-purpose language for defining QoS properties. It allows both interface annotation as well as component type annotation. Moreover, it has support for UML integration based on a lightweight QoS profile and has QoS negotiation capabilities. All the previous work on QoS specification languages including QML [7] (QoS Modeling Language) and QuO [19] (Quality Objects) is superseded by [18].

Therefore, we limit our comparison of QoS specification languages to the quality modeling language developed by Aagedal. Our CQML has been designed to be superimposed on domain specific component-based system composition modeling languages and not with interface definition languages as in the case of Aagedal's QoS language. The latter allows QoS annotations at type level (IDL interface and component definition) only and therefore, cannot be used to specify QoS requirements on components on a per-instance basis. Although, the QoS specification capability in our CQML is not as general as in Aagedal's quality modeling language, instance level QoS specification is possible in our CQML.

Lightweight and heavyweight extensions for UML are possible to create QoS profiles using extensibility mechanisms provided by UML. Lightweight extensions use only the mechanism of stereotypes, tagged values, and constraints. Heavyweight extensions require modification to the UML metamodel, which is naturally more intrusive than lightweight approaches. The OMG has adopted UML profile [11] for schedulability, performance and time specification, which is based on lightweight extensibility mechanisms of UML. OMG has also adopted a more general profile for modeling QoS [13]. This UML profile provides a way to specify the QoS ontology with QoS characteristics. It has support for attaching QoS requirements to UML activity diagrams. A common feature between these UML profiles and CQML is that both have first class support for QoS concerns. Compared to the lightweight mechanisms of above-mentioned UML profiles, CQML requires heavyweight metamodel level composition of two languages. A benefit of this approach is that the full strength of the metaprogramming environment can be leveraged in the process.

Another approach [6] for managing QoS is based on the QuO framework. It is an aspect-based approach to programming QoS adaptive applications that separates the QoS and adaptation concerns from the functional and distribution concerns. It puts more emphasis on encapsulating the system adaptation and interactions as an aspect. But it is more applicable to the CORBA-based platforms. The work described in [8] shows similarities between network-level configurable protocols and aspects. Both of the above mentioned approaches focus on lower-level OS and network related QoS whereas CQML is an AOM approach that focuses on the higher level platform-independent QoS concerns in component based systems and provides intuitive, visual abstractions. These lower-level concerns can be modeled as separate declarative QoS aspects in CQML.

5 Concluding Remarks

Large-scale component-based systems often incur secondary non-functional concerns comprising quality of service (QoS), which crosscut the system's primary concern: structural composition. The scattering and tangling of these secondary concerns impede comprehensibility, reusability and evolution of component-based systems. A Domain-specific Modeling (DSM)-based approach holds promise to address these challenges because it raises the level of abstraction at which the systems are designed and reasoned about. The complexity of system design incurred due to the crosscutting concerns, however, is not eliminated even at a higher level of abstraction because of lack of the right DSM-level modularizing abstractions.

The key contribution of this paper is to address the challenges in DSM tools for component-based system development. We described a reusable, platform-independent Component QoS Modeling Language (CQML) which when composed with a host structural modeling language, enhances the host language with declarative QoS modeling capability without breaking the syntax, semantics and the tool support of the host language. CQML not only provides separate views to specify and manipulate QoS concerns and system's structural concerns but also allows QoS metamodel to be extended without affecting the structural modeling capabilities of the host language. We evaluated this capability of CQML by composing it with a rich structural modeling language: PICML [2]. Finally, we also demonstrated how CQML's fault-tolerance models can be used to automatically generate structural and deployment models of fault monitoring infrastructure using an aspect-oriented model weaver.

The capabilities of CQML are available in open source from the CoSMIC tool web site at `www.dre.vanderbilt.edu/cosmic`.

References

[1] Jaiganesh Balasubramanian, Sumant Tambe, Balakrishnan Dasarathy, Shrirang Gadgil, Frederick Porter, Aniruddha Gokhale, and Douglas C. Schmidt. NetQoPE: A Model-Driven Network QoS Provisioning Engine for Enterprise Distributed Real-time and Embedded Systems. In *Proceedings of the 14th IEEE Real-time and Embedded Technology and Applica-*

tions Symposium, pages 113–122, St. Louis, MO, USA, April 2008.

[2] Krishnakumar Balasubramanian. *Model-Driven Engineering of Component-based Distributed, Real-time and Embedded Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, September 2007.

[3] Clemens Szyperski. *Component Software — Beyond Object-Oriented Programming - Second Edition*. Addison-Wesley, Reading, Massachusetts, 2002.

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

[5] Dionisio de Niz, Gaurav Bhatia, and Raj Rajkumar. Model-Based Development of Embedded Systems: The SysWeaver Approach. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 231–242, Washington, DC, USA, August 2006. IEEE Computer Society.

[6] Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building Adaptive Distributed Applications with Middleware and Aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 66–73, New York, NY, USA, 2004. ACM Press.

[7] Svend Frolund and Jari Koistinen. Quality of Service Specification in Distributed Object Systems. *IEE/BCS Distributed Systems Engineering Journal*, 5:179–202, December 1998.

[8] Matti Hiltunen, François Taïani, and Richard Schlichting. Reflections on Aspects and Configurable Protocols. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2006. ACM Press.

[9] Gabor Karsai, Sandeep Neema, Ben Abbott, and David Sharp. A Modeling Language and Its Supporting Tools for Avionics Systems. In *Proceedings of 21st Digital Avionics Systems Conference*, Los Alamitos, CA, August 2002. IEEE Computer Society.

[10] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgysei, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[11] Object Management Group. *UML Profile for Schedulability, Performance, and Time Specification*, Final Adopted Specification ptc/02-03-02 edition, March 2002.

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

[13] Object Management Group. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Joint Revised Submission*, OMG Document realtime/03-05-02 edition, May 2003.

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

[15] Software Composition and Modeling (Softcom) Laboratory. Constraint-Specification Aspect Weaver (C-SAW). www.cis.uab.edu/ gray/Research/C-SAW, University of Alabama at Birmingham, Birmingham, AL.

[16] Sumant Tambe, Jaiganesh Balasubramanian, Aniruddha Gokhale, and Thomas Damiano. MDDPro: Model-Driven Dependability Provisioning in Enterprise Distributed Real-Time and Embedded Systems. In *Proceedings of the International Service Availability Symposium (ISAS)*, pages 127–144, Durham, New Hampshire, USA, 2007.

[17] Jules White, Douglas C. Schmidt, and Aniruddha Gokhale. Simplifying autonomic enterprise java bean applications via model-driven engineering and simulation. *Journal of Software and System Modeling*, 7(1):3–23, 2008.

[18] Jan Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, Oslo, March 2001.

[19] John A. Zinky, David E. Bakken, and Richard Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.