CQML: Aspect-oriented Modeling for Modularizing and Weaving QoS Concerns in Component-based Systems

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

Abstract

Current domain-specific modeling (DSM) frameworks for designing component-based systems often consider the system's structural and behavioral concerns as the two dominant concerns of decomposition while treating non-functional or quality of service (QoS) concerns as an after thought. Such frameworks lack a 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 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 aspect-oriented modeling approach that provides strong separation between the structural and non-functional dimensions. CQML supports independent evolution of structural as well as QoS metamodel of composition modeling languages. The join point model of CQML enables declarative QoS aspect modeling and supports automatic weaving of structural changes effected by QoS requirements. We evaluate the capabilities of CQML for a variety of structural modeling languages and provide quantitative results indicating the modeling effort saved in automating the weaving of QoS concerns.

1 Introduction

Recent advances in domain-specific modeling (DSM) [19] have resulted in numerous DSM tool suites for designing large, component-based software systems with multiple quality of service (QoS) requirements (*e.g.,* predictable latencies, fault-tolerance, and security.) Recent successes with DSM tools in this area include the Embedded Systems Modeling Language (ESML) [13] for avionics mission computing, SysWeaver [7] for embedded systems, and our earlier work on the Platform Independent Component Modeling Language (PICML) [3] for a range of distributed, real-time systems. These DSM tools provide support for component-based software engineering (CBSE) [5] wherein systems can be modeled by composing different components, each encapsulating a reusable unit of functionality.

Despite the number of benefits of these DSM tools and techniques, 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 simultaneously satisfied along with the system's functional composition (*i.e.*, primary) concerns.

As an example of a non-functional requirement that affects system's the 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, across a group of components, and across nested component groups. Replication, which is the most widely used technique for developing highly available systems, inherently requires additional copies of components that must be composed with the original business components. This illustrates the scattering of the fault-tolerance concern across the system's functional composition dimension. Scattering of fault-tolerance concerns is also observed along other non-functional concerns, such as deployment planning because application deployment must now account for the placement of the replicated business components, proxies, and application liveness monitoring components along with original business components.

To assist in designing systems where non-functional concerns crosscut with structural concerns, DSM tools must provide strong decoupling between the system's structural and non-functional concerns and must combine them when the final system is realized. Such decoupling should not only provide different views for different concerns (*view-per-concern*) but should also enable independent evolution of the modeling capabilities of each view. Evolution of the modeling capabilities of a concern view often requires en-

hancements to the metamodel of the view. Supporting independent evolution of metamodels of each concern views shortens the development lifecycle by allowing parallel enhancements to the modeling capabilities (*i.e.*, the metamodel) 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. Despite QoS being a platform-agnostic concept, DSM tools tend to tightly couple QoS with the structural characteristics. However, the 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. However, contemporary DSM tools are based on ad-hoc designs of metamodels for modeling QoS that couple them tightly with structural capabilities, preventing their reuse in other component platforms and limiting extensibility.

This paper describes our solution to address these limitations of DSM design tools for CBSE. We present Component QoS Modeling Language (CQML), which is a reusable, aspect-oriented modeling (AOM) [10] framework developed using the Generic Modeling Environment (GME) [1]. CQML is designed to be superimposed on a wide range of structural composition modeling languages as long as they conform to a small set of invariant structural properties defined by CQML. Based on these invariant properties, CQML defines an abstract join point [14] model for associating QoS aspects to the structural elements. The join point model defines where the QoS aspects meet structural elements.

Around its abstract join point model, CQML has an extensible QoS modeling framework that allows declarative QoS requirements to be associated with structural component models. The QoS requirements are modularized using what we call *declarative QoS aspects.* They bind the QoS advice to the join points of the underlying structural modeling language. In this paper we demonstrate how abstract syntax of purely structural modeling languages can be *retroactively* enhanced with the QoS modeling capabilities of CQML by superimposing CQML's join point model. We evaluate this capability using three different structural modeling languages for component-based systems.

Based on CQML's QoS annotation capabilities, we propose an analysis framework with a round-tripping mechanism to populate the results of the analysis back into the original models. The semantics of declarative QoS advices are handled in the analysis framework. The framework leverages CQML's abstract join point model and a novel technique called *metamodel reflection* to read and update structural models built using different languages without knowing all the details about their abstract syntax.

We evaluate capabilities of CQML by illustrating an example of an availability QoS advice. We demonstrate how application models built using different structural languages annotated with availability QoS advice can be analyzed using our deployment planning tool, which is a part of the analysis framework. Finally, we show how structural manipulations effected by the analysis can be woven back into the original models automatically.

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) [1] toolkit. CQML strongly separates the system's QoS concerns from the structural composition concerns. It also supports customizable QoS modeling for multitude of component-based platforms depending upon their structural characteristics. Figure 1 shows the process of using CQML. We now describe how CQML uses this process to resolve the challenges described before.

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) [13] for embedded systems, J2EEML [22] for Enterprise Java Beans, and Platform Independent Component Modeling Language (PICML) [3] for Light-weight CORBA Component Model (LwCCM) [16] exist today that

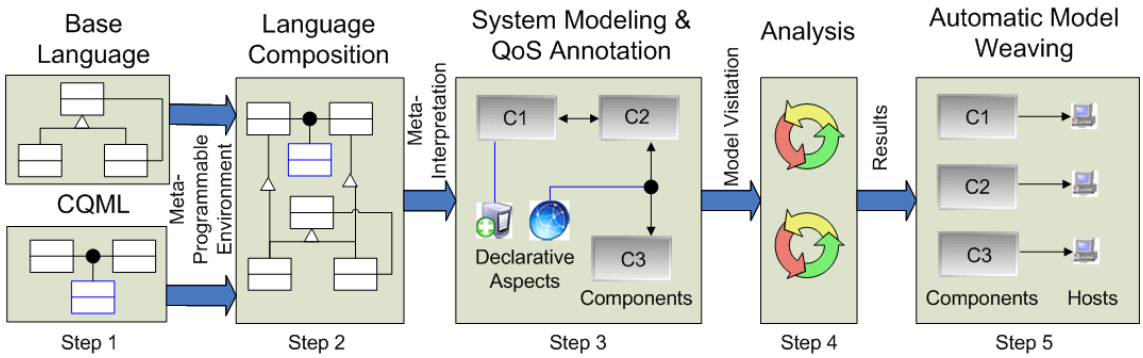

Figure 1: Process Model for Reusing CQML for QoS Modularization and Weaving

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.

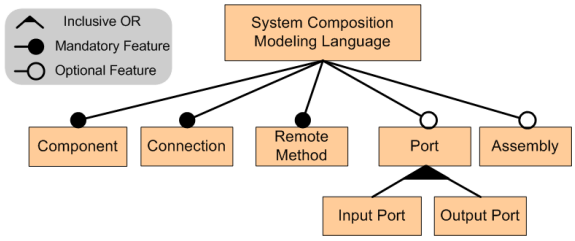

Figure 2: A Feature Model of Composition Modeling Language

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 [6] 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, metamodel composition [4] techniques are used to *mixin* the metamodel of CQML with that of the base composition modeling language producing a composite language, which has the capabilities of both the constituent languages. In step 3, the composite language is used to model component-based systems with QoS aspect modeling capabilities of CQML. In step 4, system models annotated with CQML QoS aspects are used for analyses without having to deal with the specifics of the underlying base language. Finally, in step 5, system models are populated automatically with the results of analysis, if needed. We describe step 4 and 5 in detail in section 2.5.

2.2 Extensible Design of CQML

Based on the feature model of component-based modeling languages, CQML builds an extensible QoS modeling layer. CQML associates declarative QoS aspects to one or more of the invariant properties of the underlying base language. We have designed several declarative QoS aspects that are applicable to a general class of component-based systems. We have developed (1) *FailOverUnit* [20], which modularizes fault-tolerance requirements of components and assemblies, (2) *SecurityQoS*, which modularizes role-based access control policies of port based communication between components, and (3) *NetworkQoS* [2], 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 [2] and [20] to read more on *NetworkQoS* and *FailOverUnit* modeling respectively.

To support extensions of QoS metamodel, CQML defines a set of abstract QoS elements: *Component-QoS*, *Connection-QoS*, *Port-QoS*, *Assembly-QoS* and *Method-QoS*. CQML can be extended with new concrete declarative QoS modeling capabilities by inheriting from these basic abstract QoS elements. To enhance CQML with a concrete QoS aspect, a language designer has to extend the metamodel of CQML at the well-defined points of extension represented by the five abstract QoS elements. By doing so, the concrete modeling aspects inherit the (1) abstract syntax, (2) associations, (3) cardinality constraints, and (4) visualization constraints of the abstract QoS entities. For example, as shown in Figure 4, *FailOverUnit* inherits as-

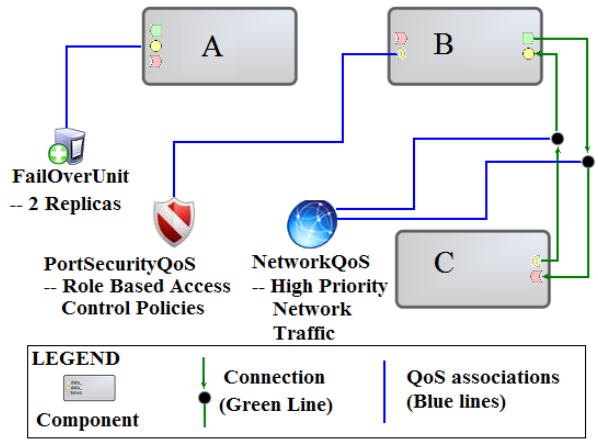

Figure 3: Declarative QoS Aspect Modeling Capability of CQML

sociation constraints from the abstract *ComponentQoS* and *AssemblyQoS*. Therefore, *FailOverUnit* can be associated with components and assemblies only and never with ports or connections.

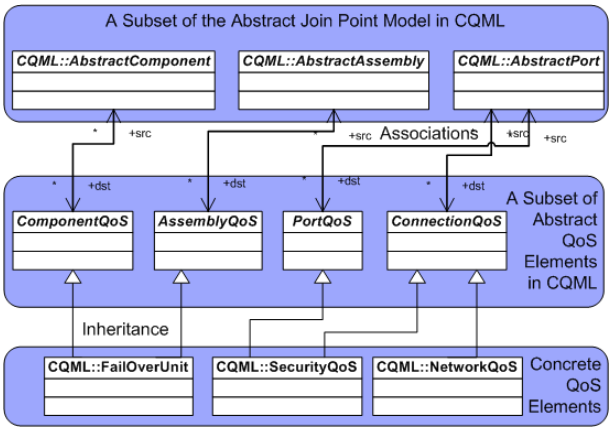

Figure 4: Simplified Meta Model of CQML

Separation of structural concerns and QoS concerns is achieved using separate views for QoS and structural elements (*view-per-concern*). The visibility of concrete QoS modeling elements is controlled 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. These constraints are inherited by all the concrete QoS elements that are derived from one or more abstract QoS elements as shown in Figure 4. Due to inheritance of these constraints, the concrete QoS elements are also projected and manipulated in the QoS view. Thus CQML metamodel not only provides QoS modeling capability, it does so while achieving separation of concerns at modeling level.

2.3 An Abstract Join Point Model for Component Modeling Languages

Along with the abstract QoS elements in the previous section, CQML defines an abstract representation of the mandatory and optional features of a generic structural modeling language. For example, CQML defines *AbstractComponent*, *AbstractConnection*, *AbstractMethod*, *AbstractPort*, and *AbstractAssembly*. These abstract types do not have semantics of their own except being able to associate QoS aspects with them. Moreover, the abstract nature stems from the fact that they cannot exist without a concrete instantiation in the underlying base modeling language. In the following section we describe how a concrete instantiation is done using a technique called metamodel composition.

2.4 Instantiating Abstract Join Point Model Using A Concrete Structural Modeling Language

CQML's support for QoS aspect modeling can be superimposed on a structural modeling language by composing the metamodel of CQML with the metamodel of the base language to create a composite language as described by Step 2 in Figure 1. Domain abstractions in the base language such as component, assembly, port inherit from the corresponding abstract elements in CQML. Due to such inheritance, the domain abstractions in the base language inherit all the QoS related associations and constraints from CQML elements. Models of the new composite language can not only use the associations defined in the original language but also the associations inherited from CQML. Thus, all the concrete QoS aspects and their constraints are *mixed-in* with the underlying structural modeling language.

Note that the abstract join point model achieves a strong decoupling between structural and CQML metamodels. The structural metamodel of the base language can be enhanced without affecting the CQML metamodel and vice-versa. Therefore, the abstract join point model is the key to support independent evolution of the structural as well as CQML metamodel. Moreover, using the abstract join point model, multiple composite languages can be created by composing CQML with different structural modeling languages using the same process. Figure 5 shows an example of how CQML is composed with PICML to create a composite language using inheritance mechanism. After composition, PICML's *component* and *assembly* can be associated with everything that CQML's *abstract component* and *abstract assembly* can be associated with (*e.g., FailOverUnit*).

An important benefit of our approach is that CQML introduces QoS modeling capability in a base language with-

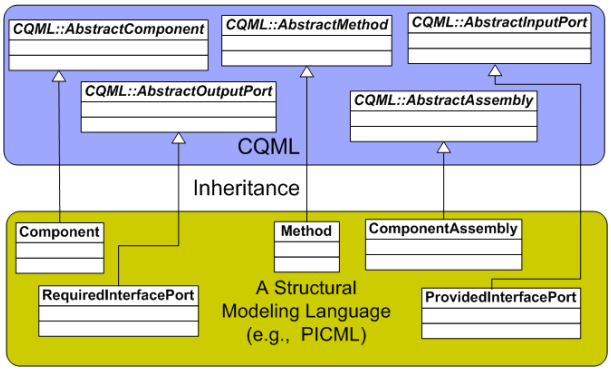

Figure 5: Composing CQML's Abstract Component Model With A Base Language Using Inheritance

out affecting its original syntax and semantics. CQML can be composed flexibly with the underlying base language even though it does not support some optional features shown in Figure 2. Using CQML with a base language that supports less number of primitives gives rise to a smaller concrete QoS model. On the other hand, composing CQML with a base language with all the mandatory as well as optional primitives gives rise to a larger QoS model.

In Section 3 we show how CQML is composed with three different base languages (PICML, J2EEML, and ESML) that have different structural modeling capabilities. Composing CQML with them gives rise to different QoS modeling capabilities in each composite language: PICML′, J2EEML′, and ESML′. Reuse promoted by CQML's generic QoS entities and its design thus lends itself to easier development of component-based systems modeling languages with QoS support. It reduces the need of reinventing previously designed artifacts for every new QoS aspect that is added.

2.5 A Framework for Developing QoS Analyses and Automated Weaving of QoS Advice

The intent of our analysis framework is not to define new techniques for analyzing component-based systems, but to support them. Hence, we leverage the join point model of CQML and the ability to associate declarative QoS aspects to the structural elements to conduct base language independent analyses of the structural properties of the system. Based on structural properties, several different analyses such as component collocation optimization [3], component workflow monitoring [21], and deployment planning [20] are possible. The analysis phase is represented by Step 4 in Figure 1.

The analyses are developed in a fashion that is independent of the underlying base language as CQML provides the necessary indirection (layer of abstraction) between the analysis tools and the actual platform-specific component model of the base language. CQML hides away the platform specific aspects of the component model and gives a clean, simplified way of associating and accessing declarative QoS aspect information from the models.

Different kinds of artifacts are possible from the analysis phase. Some analysis results are for human consumption whereas some other analysis artifacts lead to structural manipulation of the original model so that the analysis results can be processed using other tools. For example, deployment planning shown in [20] not only generates a placement for components in the system, but also requires a human to add new replica components in the original model so that a complete fault-tolerant system can be deployed. Unless an automated process is defined to populate the analysis results back into the model, performing model manipulations manually after analyses quickly becomes a time consuming and tedious task in the software development process. Therefore, an automated support to perform model manipulations is necessary after the analysis phase. Therefore, we have developed an aspect weaver framework that provides a way to populate the results, if any, of the analysis back into the model. We use the Constraint-Specification Aspect Weaver (C-SAW) [11] and Embedded Constraints Language (ECL) to populate the model with the result of the analysis tool.

In Section 3 we show how we have used our deployment planning [20] tool that takes into account the availability aspect (*FailOverUnit*) modeled using CQML to generate a placement for the components that meets the availability requirements. After the analysis, we also generate ECL code to automatically populate the replica components and their interconnection in the model which would have otherwise done manually.

2.5.1 Architecture of the Aspect Weaver Framework

Figure 6 shows the overall architecture of our aspect weaver framework. It consists of four main subcomponents: (1) *instance search engine*, (2) *QoS aspect visitor*, (3) *analysis component*, and (4) *ECL code generator*. The instance search engine is a generic component, which can be used across multiple analyses, however, the remaining three components are specific to every QoS aspect and the analysis performed on it.

Instance Search Engine. It searches and collects the instances of the concrete structural elements such as components, assemblies, and ports in a model, irrespective of the underlying base language. These structural elements are instances of the types defined in the metamodel of the composite language. The instance search engine depends on the fact that the elements that it searches are the instances of the types that specialize the abstract elements defined in CQML. The output of the instance search engine is a set of instances of abstract structural components. It filters out the base language specific type information from the collected

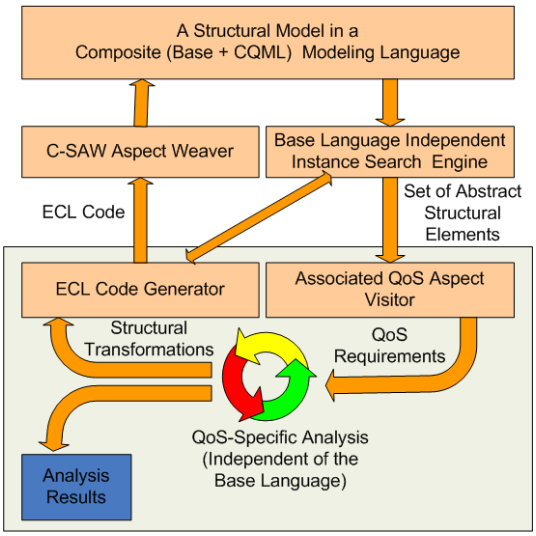

Figure 6: **Architecture of CQML-based Aspect Weaver Framework**

components before passing them to the next stage.

QoS Aspect Visitor. The QoS aspect visitor obtains the abstracted set of instances of the structural elements from the instance search engine. It then traverses the declarative QoS aspects defined in the model. The traversals are based on the abstract syntax and the associations defined in CQML. The associations defined in CQML are inherited by the base language using the metamodel composition technique described in Section 2.4. Therefore, generic traversals can be written that remain unaffected by the base language specific details of the structural elements. A key characteristic of the QoS aspect visitor is that it can be reused across multiple base language models as QoS associations inherited from CQML remain consistent across multiple composite languages.

Analysis Component. QoS-specific analysis algorithms are implemented in the analysis component. The abstract set of structural elements and declarative QoS aspects processed by the earlier two stages are available for analysis. In Section 3.2, we show how our deployment planning analysis [20] can be integrated with the aspect weaver framework.

ECL Code Generator. As the name suggests, it is used to generate ECL code that populates the model with the result of the analysis. The ECL code modularizes model manipulations that might be necessary in the later stages of system lifecycle. The generated ECL code automatically carries out the necessary model manipulations without any user intervention. This eliminates the need for the user to learn ECL and reduces modeling efforts.

3 Evaluating CQML

This section describes our evaluation of CQML. First, we demonstrate how purely structural modeling languages can be enhanced with QoS annotation capabilities by composing them with CQML. We show this capability with three different component-based structural modeling languages. Second, we evaluate CQML's QoS analysis framework by developing a deployment planning analysis for structural models of component-based systems irrespective of their underlying base language. Finally, we demonstrate how our deployment planner generates ECL code to automate model manipulations to significantly reduce human modeling efforts.

3.1 Composability of CQML with Structural Modeling Languages

To evaluate composability of CQML we chose three component-based structural composition languages: the Platform Independent Component Modeling Language (PICML) [3] for Light-weight CORBA Component Model (LW-CCM) [16], J2EEML [22] for Enterprise Java Beans (EJB), and the Embedded Systems Modeling Language (ESML) [13] for embedded systems.

There are many commonalities and differences among these languages that stem from the differences in the underlying component model that they model. Table 1 summarizes the similarities and the differences between these three languages. All of them are component-based system modeling languages, which treat components as first class entities and have varying degree of support for assemblies (nesting of components and assemblies.) For example, J2EEML and PICML support hierarchical composition of assemblies but ESML has a flat, single level structure of components. All the three languages support the notion of a connection. The notion of provided interfaces (an implementation of a particular interface) is present in PICML and ESML but not quite explicit in J2EEML. It manifests itself in a weaker form of just a set of invocable methods on a bean. Similarly, the notion of required interfaces[1] is present in PICML and ESML but is absent in EJB and hence in J2EEML. In summary, PICML′ feature set turns out to be a super-set of the features of the other two languages.

Using specializations to the join point model, we composed CQML with the above three languages giving rise to three composite languages: PICML′, J2EEML′, and ESML′. The concrete join point model of the three composite languages varies because of the varying structural capabilities of the underlying base languages. The richness of the join point model determines the ability of the composite language to attach declarative QoS aspect to the structural elements in a model.

[1] It describes an ability of a component to use an interface implementation supplied by some external component.

Supported Features	PICML	J2EEML	ESML
Component, Methods, and Connections	Yes	Yes	Yes
Provided Interface Ports	Yes	No	Yes
Required Interface Ports	Yes	No	Yes
Assemblies	Yes	Yes	No

Table 1: **Comparison of Capabilities of Selected Three Modeling Languages**

Structural Elements	PICML′	J2EEML′	ESML′
Component	FailOverUnit	FailOverUnit	FailOverUnit
Assembly	FailOverUnit	FailOverUnit	N.A.
Connections	NetworkQoS	NetworkQoS	NetworkQoS
Provided Interface Ports	SecurityQoS	N.A.	SecurityQoS
Required Interface Ports	SecurityQoS	N.A.	SecurityQoS

Table 2: **Enhanced QoS Aspect Modeling Capabilities of Composite Languages PICML′, J2EEML′, and ESML′**

Table 2 summarizes the enhanced QoS aspect modeling capabilities of the composite languages. All three composite languages had new support for modeling *FailOverUnit*s, which are associated with components. However, J2EEML′ could not support the QoS advice association with required interface like PICML′ and ESML′ could because there is no support for ports built in to J2EEML. Similarly, in ESML, *FailOverUnit* cannot be associated with assemblies because assemblies is not supported.

All the above QoS modeling enhancements are projected into and manipulated from the *QoS* view. This graphical view of the model is separate from *structural* view where hierarchical systems are composed using components. This feature provides visual separation of structural concerns from QoS concerns. Moreover, the metamodel of the structural view and CQML can be enhanced in parallel, if needed, and can be composed again as shown in Figure 1.

The results indicate that CQML can be composed with a variety of component-based structural composition languages to introduce QoS modeling support in them while supporting strong separation and independent evolution of QoS and structural concerns.

3.2 Evaluating CQML's Analysis Framework and Automatic Model Weaving Support

In this section we demonstrate how base language independent QoS analysis can be conducted using CQML's analysis framework. Moreover, we also evaluate savings in human modeling efforts due to automatic weaving of modeling artifacts generated by QoS analysis. We leverage our previous work on availability analysis called MDDPro [20].

We developed a variant of our MDDPro deployment planning tool to evaluate the modeling and automation capabilities of CQML. Based on the availability concerns that are captured using *FailOverUnit*, our variant of MDDPro (1) generates replicas of the components annotated with FailOveUnit and (2) runs a placement planner on the components and their replicas to decide a placement. The planner is based on the shared risk group (SRG) [20] hierarchy of the hosts in a domain that allows us to place component replicas in a way that minimizes the risk of simultaneous failure of replicated functionality. MDDPro also allows plugging in different replica placement algorithms to improve system availability.

To evaluate the framework, we developed a prototype component-based application for an aircraft *GPS auto-navigator.* Our GPS auto-navigator has four components as shown in Figure 7: `Timer`, `GPS`, `AirFrame` and `NavDisplay`. The Timer component sends a periodic message to the `GPS` component in response to which the `GPS` component updates its location information from satellite and sends it to the `AirFrame` component. The `AirFrame` component adjusts the path of the aircraft, if necessary, and sends the current location information to the `NavDisplay` component, which renders it on a graphical device for human consumption. A shared risk group (SRG) hierarchy of four hosts was created where the system is to be deployed.

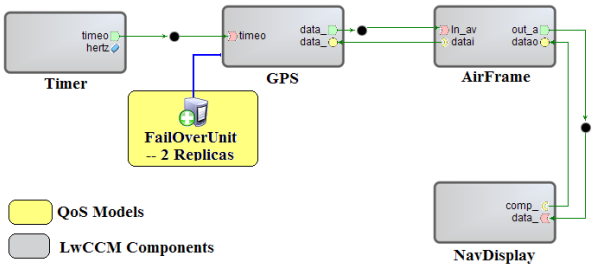

Figure 7: **A Model of Aircraft GPS Auto-navigator in PICML′**

We created three structurally identical models of the GPS navigator in three composite languages: PICML′, J2EEML′, and ESML′. The model in PICML′ is shown in Figure 7. We annotated the `GPS` component in all three models identically using CQML's availability QoS aspect model, *FailOverUnit*. We specified that 2 replicas of the GPS component be made as shown in Figure 7. The deployment planner of MDDPro visits the availability models that are attached to the `GPS` component. Based on the component replication degree and the given shared risk group hierarchy [20], the planner generated placement for the four primary components as well as the two replicas of `GPS` in all three models. We verified that the output of the plan-

# of Replica	Generated ECL LOC	FailOverUnit associated with GPS Component			Generated ECL LOC	FailOverUnit associated with GPS, AirFrame, and NavDisplay Components		
		Comp.	Conn.	Ports		Component	Connection	Ports
1	52	1	3	3	114	3	12	9
2	74	2	6	6	246	6	32	18
3	96	3	9	9	426	9	60	27

Table 3: **Savings in Modeling Effort of Components, Connections, and Ports due to Automatic Generation**

ner, which is a simple component to physical host mapping is identical for all three models of the composite languages. Our results indicate that deployment planning can be done in a base language independent way using CQML tool-suite.

Even though base language independent analysis can be done using CQML's tool-suite, some artifacts generated by the deployment planner such as replica components must be populated back into the original model so that appropriate platform-specific descriptors (*e.g.,* packing and deployment metadata in XML) can be generated from the model in the later stages of system lifecycle. Creating such replica components manually is a tedious and error prone task and does not scale well as the size of the system model increases. Therefore, we developed a generator that executes after the deployment planner and synthesizes ECL code that weaves the availability QoS aspect in the original system model with replica components and their interconnections.

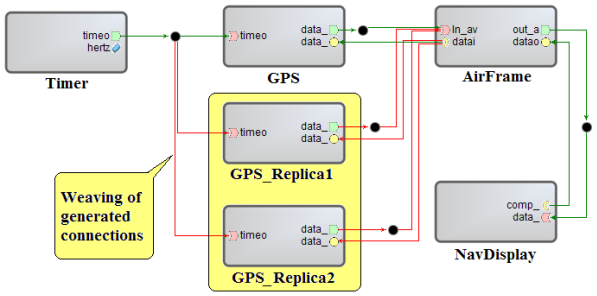

Figure 8: **Result of Weaving Generated Components and Connection in PICML′**

The generated ECL code performs the following steps to weave the availability QoS aspect. First, it creates multiple identical copies (clones) of the components and assemblies that are annotated with *FailOverUnit* QoS aspect. The number of clones are dictated by the degree of replication captured using *FailOverUnit*. Figure 8 shows `GPS_Replica1` and `GPS_Replica2` which are clones of the `GPS` component to which *FailOverUnit* was associated in the original model. Second, the ECL code recreates the same connections for the replica components as that of the original `GPS` component. For example, the `GPS` component has connections with the `Timer` and the `AirFrame` component, which are recreated between the `GPS_Replica1` and `GPS_Replica2` components. Figure 8 shows the final result of weaving of generated components and connections in the aircraft GPS auto-navigator model in PICML′ language.

Table 3 summarizes the savings in efforts due to automation provided by the CQML tool-suite. The table shows how much modeling effort is saved by generated ECL code if (1) only the `GPS` component has *FailOverUnit* associated with it and (2) `GPS`, `AirFrame`, and `NavDisplay` have a *FailOverUnit* associated with them. Savings in the manual efforts in the second case are much more significant because the number of connections between components grows multiplicatively when replication degree of components increases linearly. Note that Figure 8 shows only the first case.

It is clear from the table that without automatic ECL code generation capability of CQML tool-suite, the modeler would have to manually create the components and connections between them. Moreover, the modeler also must take deployment decisions of the replicated components if the deployment planner is not used. ECL code generator produces necessary aspect weaving code for C-SAW to execute and thereby eliminating the manual steps.

4 Related Work

Capturing QoS specifications at design-time has long been a goal of researchers [9, 23]. A prior effort, called Component Quality Modeling Language [23], 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 [9] (QoS Modeling Language) and QuO [24] (Quality Objects) is superseded by [23].

Therefore, we limit our comparison with the QoS specification language developed by Aagedal. 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 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 [15] 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 [17]. 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 (*e.g.,* GME) can be leveraged in the process.

The SysWeaver [7] approach is a MDE-based technique for developing real-time systems. It supports design-time timing behavior verification of real-time systems and also supports automatic code generation and weaving for multiple target platforms. In the SysWeaver approach, there is an explicit step where the system functional model specified in Simulink must be translated into SysWeaver model to perform different analyses. On the other hand, we eliminate the need for transformation of platform-specific system functionality models into analysis domain models. We expect great savings in manual efforts where such automatic transformations are not provided or possible. Moreover, SysWeaver does not address tangling of availability concerns into structural concerns. The replicas of protected components need to be explicitly modeled in the functional view of the Simulink model.

Object Management Group (OMG) has developed an extension to Unified Modeling Language (UML) for Modeling and Analysis of Real-time and Embedded systems (MARTE) [18]. MARTE profile provides foundations for model-based descriptions of real time and embedded systems and provides facilities to annotate UML models with information required to perform specific analyses such as, performance and schedulability analysis. MARTE profile has provisions for a generic component model as well. CQML and MARTE profile both share common design goals such as annotating application models with information necessary for performing multiple analyses on models. However, MARTE considers UML as its only underlying modeling language where as CQML's modeling capabilities can be superimposed on multiple DSMLs provided a small set of invariants are satisfied by them. Contrary to CQML, the analysis framework of MARTE profile does not propose any round-tripping mechanism to populate the results of analysis back into the original models.

Another approach [8] 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 [12] 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 aspect-oriented modeling 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) that defines an abstract join point model which when composed with a base 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 three structural modeling languages for component-based systems: Platform Independent Component Modeling Language (PICML) [3], J2EEML [22], and Embedded Systems Modeling Language (ESML) [13]. Finally, we also demonstrated how CQML's availability QoS aspects can be used to automatically generate placement decisions using our deployment planner and how replica components and their interconnections be weaved into the original model 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] Ákos Lédeczi, Árpád Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[2] J. Balasubramanian, S. Tambe, B. Dasarathy, S. Gadgil, F. Porter, A. Gokhale, and D. C. Schmidt. Netqope: A model-driven network qos provisioning engine for distributed real-time and embedded systems. In *RTAS' 08: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 113–122, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[3] K. 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, Sept. 2007.

[4] K. Balasubramanian, D. C. Schmidt, Z. Molnar, and A. Ledeczi. Component-based system integration via (meta)model composition. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 93–102, Washington, DC, USA, 2007. IEEE Computer Society.

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

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

[7] D. de Niz, G. Bhatia, and R. Rajkumar. Model-based development of embedded systems: The sysweaver approach. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 231–242, Washington, DC, USA, 2006. IEEE Computer Society.

[8] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. 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.

[9] S. Frolund and J. Koistinen. Quality of Service Specification in Distributed Object Systems. *IEE/BCS Distributed Systems Engineering Journal*, 5:179–202, Dec. 1998.

[10] J. Gray, T. Bapty, and S. Neema. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, pages 87–93, October 2001.

[11] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan. An Approach for Supporting Aspect-Oriented Domain Modeling. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*, 2003.

[12] M. Hiltunen, F. Taïani, and R. 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.

[13] G. Karsai, S. Neema, B. Abbott, and D. Sharp. A Modeling Language and Its Supporting Tools for Avionics Systems. In *Proceedings of 21st Digital Avionics Systems Conference*, Los Alamitos, CA, Aug. 2002. IEEE Computer Society.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

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

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

[17] 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.

[18] Object Management Group. *UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)*, OMG Document realtime/05-02-06 edition, May 2005.

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

[20] S. Tambe, J. Balasubramanian, A. Gokhale, and T. 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.

[21] S. Tambe, A. Dabholkar, A. Gokhale, and A. Kavimandan. Towards A QoS Modeling and Modularization Framework for Component-based Systems . In *Proceedings of the EDDOC Workshop on Advances in Quality of Service Management (AQuSerM)*, September 2008.

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

[23] J. Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, Oslo, Mar. 2001.

[24] J. A. Zinky, D. E. Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.