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

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

Abstract

Designing large, component-based systems with multiple quality of service (QoS) concerns is a hard problem because these concerns crosscut the system functional composition concerns and get tangled with other para-functional concerns, such as deployment. Current model-driven engineering (MDE) system design tools tend to focus predominantly on system functional composition, and tightly couple QoS modeling concerns with structural concerns. Moreover, the tools are often component technology-specific although the notion of composition and QoS are inherently platform-independent resulting in multiple MDE tools that reinvent solutions to the same problems.

This paper describes the Component QoS Modeling Language (CQML), which is a reusable, platform-independent, aspect-oriented modeling approach for separation of crosscutting concerns for QoS properties. CQML is applicable to all those functional composition modeling languages which conform to a small set of invariant properties. The join point model of CQML enables declarative QoS aspect modeling and automated weaving of QoS concerns into the base modeling language. We evaluate the capabilities of CQML for a variety of base modeling languages and provide quantitative results indicating the modeling effort saved in automating the weaving of QoS concerns.

Categories and Subject Descriptors D:Software [*2:Software Engineering*]: 2:Design Tools and Techniques

General Terms Modeling, Crossutting, Composition

Keywords AOM, MDE, DSML, QoS, Aspects.

1. Introduction

Recent advances in model-driven engineering (MDE) [22] have resulted in MDE tool suites for designing large, component-based software systems with multiple quality of service (QoS) requirements, such as predictable latencies, availability and security. Recent successes with MDE tools in this area include the Embedded Systems Modeling Language (ESML) [11] for avionics mission computing, SysWeaver [3] for embedded systems, and our earlier work on the Platform Independent Component Modeling Language (PICML) [1] for a range of distributed, real-time systems. These MDE tools provide support for component-based software engineering (CBSE) [25] wherein systems can be modeled by composing multiple different components, each encapsulating a reusable unit of functionality.

Despite the number of benefits of these MDE tools and techniques, such as enhanced reusability and extensibility, however, designing operational QoS-intensive systems remains a significantly hard problem due to the multiple crosscutting para-functional properties (*i.e.*, the secondary concerns) that must be satisfied simultaneously along with system functional composition (*i.e.*, the primary concern). Figure 1 depicts a part of the para-functional concern space we are interested in, which includes the dimensions of QoS and deployment issues that are scattered across the primary dimension of system functional composition, and are tangled with each other.

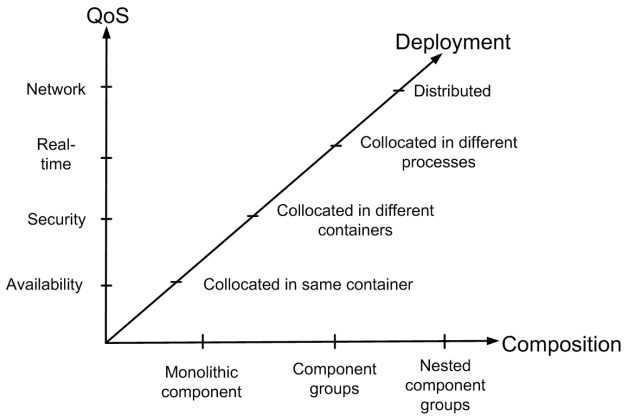

Figure 1: Para-functional Concern Space in CBSE

As a concrete example, consider how the algorithms and techniques used to enhance system availability – a QoS concern – including one or more replication schemes, such as active or passive, may have to be incorporated at different levels of granularity of system functional composition *e.g.*, on a per-component basis, across a group of components, or across nested component groups. Replication inherently requires additional functional components that must be added and composed. Depending on whether the replication scheme used is active or passive, a group communication multicast protocol with appropriate duplicate request/reply suppression capability is required, or complex state synchronization schemes are required to be added to the system functional composition dimension. This illustrates the scattering of the availability concern (and QoS in general) across the system functional composition dimension.

Scattering (or crosscutting) of availability (and in general QoS) along the primary dimension of system functional composition is not the only challenge. As depicted in Figure 1, the availability concern impacts other para-functional system concerns, such as deployment. A deployment is essentially a binding of a component or group of components to a node of the system. The impact of the availability concern on the deployment concern is also non trivial since the deployment must now account for placing the replicated functionality on the system resources such that the resulting availability of the overall system is maximized. This demonstrates the tangling of the QoS and deployment concerns.

The impact of scattering and tangling of para-functional concerns with system functional composition is non trivial for MDE design tool users *i.e.*, the system developers. First, they must reason about the entire system design in the presence of the crosscutting and tangled concerns. This is often the result of a MDE tool suffering from the *tyranny of dominant decomposition* [20], wherein the tool provides support for system composition and may also provide different views to provide a visual separation of concerns. Yet the developers must still perform the onerous and often error-prone task of adding substantial new elements to the system design to address the para-functional concerns, and that too in a non intuitive manner.

Second, these problems are compounded by the heterogeneity in the available MDE tools. The variability in the semantics and programming models of contemporary component platforms, such as the CORBA Component Model (CCM), J2EE/EJB and .NET Web Services often make the MDE tools platform-specific although they all support the notion of system composition, QoS and deployment, all of which are inherently platform-independent issues. This requires system developers, who are domain experts, to use different platform-specific MDE tools depending on the platform for which their systems are designed.

This paper describes our solution to address these two limitations of MDE design tools for CBSE. We describe the Component QoS Modeling Language (CQML), which is a reusable and platform-independent aspect oriented modeling (AOM) [7] framework developed using the Generic Modeling Environment (GME) [13]. CQML can work across a wide range of system functional composition modeling languages (or base languages) as long as they conform to a small set of invariant properties.

These invariant properties define the join point [12] model of CQML, which promotes QoS concerns to first class entities, and enables separation of concerns [20] and modularization for different QoS properties into what we call *declarative QoS aspects*. It enables the binding of QoS advice to the join points in the form of QoS aspect models composed within the base modeling language.

We evaluate CQML by illustrating an example of an availability QoS advice and how its impact on deployment planning can be woven back automatically in the existing base models. We demonstrate this capability across multiple base languages.

The remainder of this paper is organized as follows: Section 2 describes the design challenges for CQML; Section 3 describes the AOM design of CQML; Section 4 evaluates the benefit of CQML using a sample deployment planning tool for multiple composition modeling languages; Section 5 describes related research; Section 6 discusses the benefits and limitations of CQML; and Section 7 describes concluding remarks outlining lessons learned and future work.

2. CQML Design Challenges

Section 1 outlined two shortcomings of contemporary MDE tools used in CBSE. Resolving these shortcomings in a reusable, platform-independent modeling capability like CQML poses certain design challenges, which we describe below.

2.1 Challenge 1: Platform-independent Support for Declarative Aspects

It is well known [22] that MDE raises the level of abstraction at which various properties of large systems including functional and para-functional properties can be reasoned about. For component-based systems, MDE tools will typically provide intuitive abstractions for system composition as shown in Figure 2 . Although a higher level of abstraction is desirable, experience have shown that without proper support for modularization and separation of concerns, these tools suffer [7] from the *tyranny of dominant decomposition* [20]. In other words, the system composition support provided by these MDE tools is often geared toward only one dominant dimension of decomposition: the functional dimension.

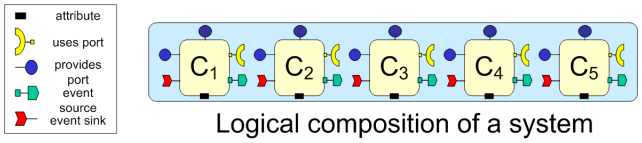

Figure 2: Software Model of System Functional Composition in CBSE

Several challenges stem from this limitation because of lack of support for decomposition along the para-functional dimensions such as QoS. For example, to assure high availability of systems, constructing models that represent placement decisions is tedious because the deployer has to make sure that availability requirements of the system are met. This is a serious case of scattering at the modeling level. The tyranny of dominant decomposition at the modeling level therefore results in the scattering and tangling of para-functional concerns as shown in Figure 3.

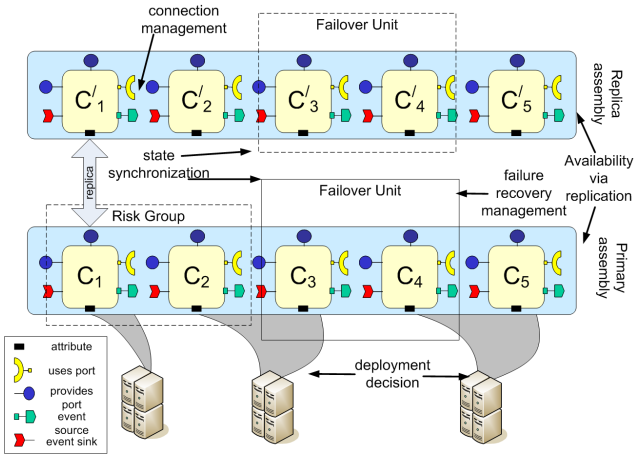

Figure 3: Scattering and Tangling of Para-functional Concerns

To address this problem, the MDE framework must provide support for modularizing different para-functional QoS concerns so that the system developers are not required to deal with the tangled system concerns that resemble the scenario depicted in Figure 3. Since CQML is meant to be an AOM capability, the modularization and subsequent weaving must be provided as a design-time capability unlike the aspects provided by AspectJ as an imperative programming capability. Thus, we are required to express *declarative QoS aspects* in a modularized way, and define a join point model [12] along the functional dimension of the system that will enable system developers to bind the declarative QoS aspect to well-defined join points in a model.

Additionally, since QoS properties are generically applicable to any component-based system, we require that CQML support this modularization at a platform-independent level yet enable choosing and activating the join points at the level of the underlying platform-specific functional composition modeling language, which we call a base language. In doing so we are required to identify the basic set of invariant properties that the range of platform-specific base modeling languages must support so that the join point model of CQML will apply to it. Section 3.1 describes how we address this challenge.

2.2 Challenge 2: Expressive Modularization of QoS Concerns

To enhance the usefulness of the MDE design tools, system designers should be able to leverage QoS modularization capabilities of CQML integrated with the MDE tool, and express the intended QoS requirements of the system using higher level intuitive, abstractions. Successive changes to the intended system QoS requirements must be accomplished at the higher level of abstraction only.

Addressing expressive modularization of QoS concerns at the modeling level requires careful design consideration. Figure 4 shows an example of a structural model of an avionics mission computing application resulting from a lack of expressive power and modularization at the modeling level.

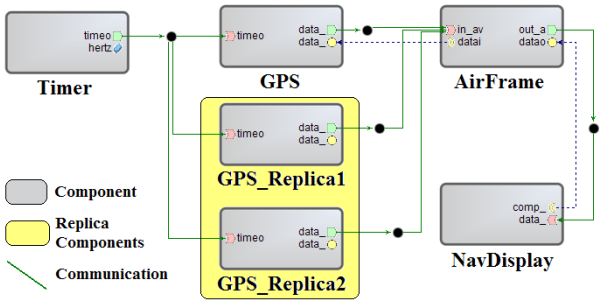

Figure 4: Scattering of Availability Concern in the Avionics Mission Computing Scenario

To overcome this problem, CQML should provide a modeler to express only the desired QoS concerns separated out from other details and using intuitive notations. Such an expressive scheme must be compatible with the join point model discussed in Section 2.1. Furthermore, system developers often want to express several different QoS properties in the system. A modeling language cannot be general enough to provide a plethora of QoS properties that may be conceived in future. CQML should therefore provide an extensible language so that the modeler can define new QoS properties rapidly. Section 3.2 describes how we address this challenge.

2.3 Challenge 3: Automated Weaving of QoS Advice

An additional challenge faced by system developers is the need to reason about various properties of the system after QoS concerns have been modeled. Consider for example that the system developer is interested in reasoning about the impact of modeling the availability concerns on the resulting deployment of the entire system shown in Figure 4. The system designer is forced to perform two actions.

First, the availability requirements modeled on the `GPS` component forces the modeler to add the two replicas of the `GPS` component in the structural view of the model. This complicates the reasoning since the availability requirements are getting tangled with the structural dimension. Second, the system developer has to determine a placement for replicas of the `GPS` component along with other business components and then model the actual association of components to nodes.

Both these actions are tedious and prone to errors. Moreover, with increasing model size, it may become infeasible to manually model these extra elements that are introduced by the QoS concerns. Ultimately, however, having such an integrated model is desired since most MDE tools provide a set of model interpreters that can synthesize different artifacts for their platforms. For example, MDE tools for component middleware platforms may provide model interpreters that synthesize the deployment and configuration metadata for their middleware platforms. This metadata is usually made up of verbose XML and is not the right abstraction at which the QoS-imposed metadata can be woven into. Therefore it is advantageous to leverage existing model interpreters in the MDE tools.

This requires that CQML should provide the means to automatically weave in QoS-specific advice into the base models. Since CQML operates at a platform-independent level, such a weaving should be feasible at this level. There exist tools like C-SAW [6] to weave in arbitrary modeling elements into existing models. However, this incurs a learning curve for system developers to understand tools like C-SAW and its language called the Embedded Constraint Language (ECL). Section 3.3 describes how we address these challenges by leveraging C-SAW but by automatically generating the desired ECL scripts corresponding to the QoS concerns.

3. CQML: An AOM Approach to Platform-independent QoS Modularization and Weaving

In this section we describe the design of the Component QoS Modeling Language (CQML), which is a platform-independent, aspect oriented modeling (AOM) 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) [13] toolkit. CQML is capable of separating QoS modeling crosscutting concerns from the primary concern of system functional composition supported by a multitude of platforms because CQML depends only on the commonalities present across the component-based systems. Some prominent examples of component platforms are CORBA Component Model (CCM) and Enterprise Java Beans (EJB). Figure 5 describes the AOM process of CQML. In the remainder of this section we describe how CQML supports this process.

3.1 Resolution 1: Platform-independent Support for Declarative QoS Aspects

We now describe how CQML resolves Challenge 1 from Section 2.1 by enabling the modularization of different para-functional QoS concerns in a platform-independent manner.

3.1.1 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. A component can either be a single indivisible unit of functionality or a collection of components assembled together as a reusable and deployable unit. Semantically rich component-based frameworks often have first class support for connectors and ports along with components. The structural (*i.e.*, functional) artifacts of a component-based system can be realized using the above-mentioned primitives in a language specifically designed for modeling system structure.

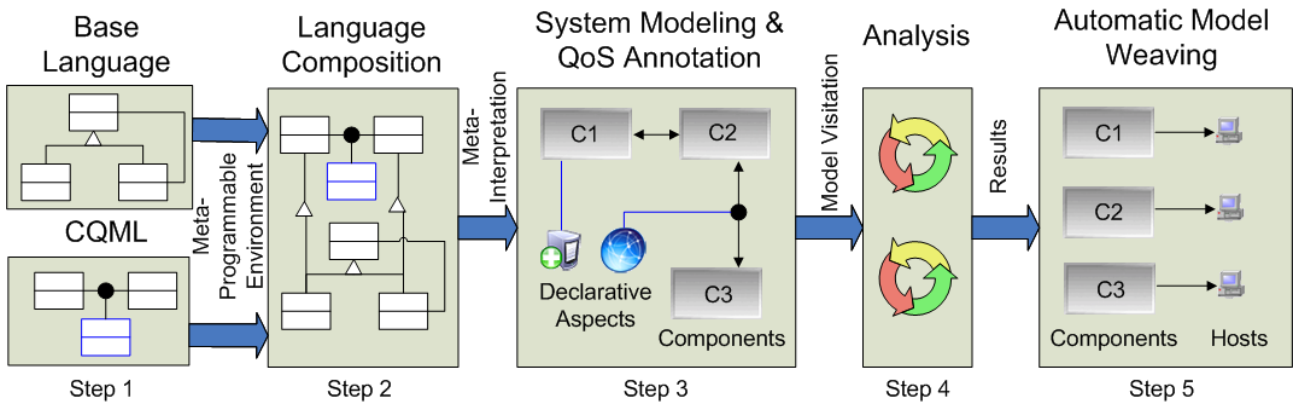

Figure 5: Process Model for Reusing CQML for QoS Modularization and Weaving

Since CQML is aimed specifically at modularizing the tangled and crosscutting para-functional properties (*i.e.,* QoS properties in our case) of component-based systems in a platform-independent manner, CQML requires an underlying base composition modeling language that allows construction and manipulation of component models. Many platform-independent as well as platform-specific component structural modeling languages, such as the Platform Independent Component Modeling Language (PICML) [1] for Lightweight CORBA Component Model [18] (LwCCM), J2EEML [27] for Enterprise Java Beans, and Embedded Systems Modeling Language [11] (ESML) for embedded systems exist today that capture various composition semantics. In this paper we have focused on languages developed in GME since CQML also uses GME though the concepts behind CQML can be applied in other tool environments.

We refer to such an underlying component modeling language as *system composition modeling language* or *base language* in short. CQML leverages the system structural modeling capability from the underlying base language. Existing as well as new base languages can be enhanced with a capability to modularize QoS concerns using CQML provided the underlying base language has a first class support for components and connectors at a minumum. The full range of modularization capabilities of CQML can be leveraged if the base language has first class support for the ports, assemblies and even deployment as described below.

Mandatory Structural Elements in the Base Language

- **Component.** A component embodies a reusable unit of functionality (either as a monolithic entity or a hierarchical assembly) that can be deployed independent of other components in the system. The base language should treat components as a first class entity.
- **Connection.** The system workflow comprising inter-operating components is captured by connections in component-based systems. The structural modeling language should therefore treat a connection as a first class entity.

Optional Structural Elements in the Base Language

- **Component Assembly** is a mechanism of composing more than one component in a hierarchical fashion. It is an important scalability feature. An assembly is an important concept since an availability concern can be bound to a group of components in a hierarchical fashion. The notion of applying a single aspect to a large part of the design is called prescriptive aspects [23].
- **Port** is an abstraction of an application-level typed communication endpoint exposed by a component or an assembly to establish one or more connections with other components. CQML also supports a finer categorization of ports into *input* and *output* ports.
- **Method** is a procedural abstraction to implement a particular business functionality. Many component-based composition modeling languages have support for modeling methods.
- **Deployment** is a platform-specific representation of metadata that encodes a mapping of components to physical nodes. The base language may or may not have its own representation of deployment. When it does not, it can borrow the deployment model in CQML and optionally extend it as long as it remains structurally compatible.

PICML, J2EEML, and ESML support all the mandatory entities mentioned above and therefore these languages can play the role of a base language for CQML as shown in Step 1 in Figure 5. In Section 4 we show that for base languages that do not support all the invariant properties mentioned above, they cannot fully leverage the declarative QoS aspect modeling capabilities of CQML.

3.1.2 Abstract Join Point Model of CQML

CQML defines an abstract join point model based on the invariant properties described earlier. The abstract form of the join point model stems from the fact that it cannot exist without a concrete instantiation of it in the underlying base modeling language. In the next section we describe how a concrete instantiation is done using a technique called metamodel composition. The invariant properties that form the abstract join point model include the first class entities namely: component, connector, port, method, and assembly. These primitives in CQML are analogous to the idea of join points in conventional AOP.

A collection of such join points is identified by the join point model. Similar to the conventional join point model in AOP, these primitives define a set of well-known points in the model to which *declarative QoS aspect* can be bound. Figure 6 illustrates the abstract join points of CQML. We call our aspects *declarative* because unlike conventional aspects, which modularize procedural statements written in a base programming language such as Java, our aspects are simple declarative annotations at the modeling level.

In CQML, the idea of pointcut expression is trivially present in the form of a simple boolean condition such as whether a particular component has declarative QoS aspect bound to it or not. It does not require the full expressive power of a pointcut expression.

3.1.3 Instantiating a Concrete Join Point Model

The abstract join point model supported by CQML makes it feasible to provide separation of crosscutting and tangled concerns at a

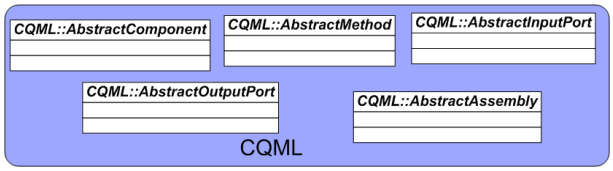

Figure 6: Abstract Join Points in CQML

platform-independent level. However, in order to realize these capabilities at the platform-specific level requires a concrete instantiation of the abstract join point model. This is realized by composing the metamodel of CQML with the metamodel of the base language to create a composed language that has the capabilities of both: the original base language and QoS modularization capability of CQML. Figure 7 shows an example of how composition of two languages can be done using a straightforward inheritance mechanism.

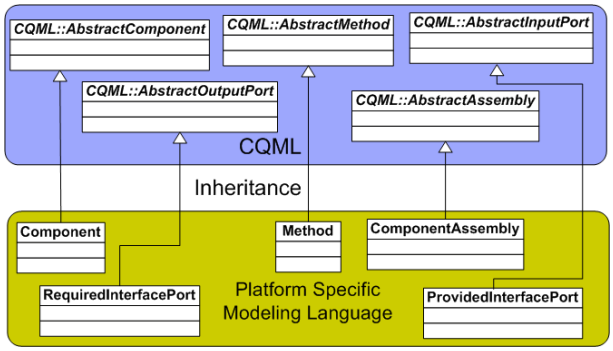

Figure 7: Instantiating Concrete Join Points using Inheritance

The real benefit of this approach comes from the fact that the abstract join point model is not limited to a specific modeling language. A different join point model can be created in another base languages if it is composed using CQML. Another important benefit of our solution is that CQML introduces a join point model in the base composition language without affecting the original syntax and semantics of the base modeling language. CQML can be composed flexibly with the underlying base modeling language even though it may not have all the basic primitives that CQML can potentially use.

Using CQML with a base language with less number of primitives gives rise to a smaller concrete joint point model. Similarly composing CQML with a language with an exhaustive set of primitives gives rise to a larger and stronger joint point model. In Section 4 we show how CQML is composed with three different base languages that have different modeling capabilities. Composing CQML with them gives rise to different joint point models in each composite language.

3.1.4 Composing CQML with a Base Modeling Language

There is a uniform approach for integrating CQML with the base language as shown in Step 2 of Figure 5 subject to the constraint that the base language support the invariant properties required by CQML. The technique of integration is based on language composition that operates at the meta level [14]. With the advent of integrated meta programming and modeling environments, such as GME [13], language composition has become practical.

A meta language is used for the specification of the abstract syntax of a DSML. It is used to precisely express concepts, relationships and integrity constraints. A model written in a meta language is called a metamodel. The metamodel of CQML is composed with the metamodel of the underlying base language to give rise to a new composed language that has the capabilities of both the languages. In evaluation section we show composition of CQML with PICML, J2EEML, and ESML giving rise to three composite languages: PICML′, J2EEML′, and ESML′.

3.1.5 Platform-independent QoS Modularization

Metamodel composition described above allows us to develop new associations between first class entities of the individual languages. To compose CQML with the base language, only one kind of relationship is required to *glue* together the two languages: Type-inheritance. CQML defines abstract types of the first class entities in the component paradigm: Method, Component, Connection, Port, Assembly. These abstract types do not have semantics of their own except being able to associate declarative QoS aspects with them. Figure 8 shows a simplified UML class diagram of the metamodel of CQML depicting how the metamodel composition gives rise to an ability to bind declarative QoS aspects to the join points.

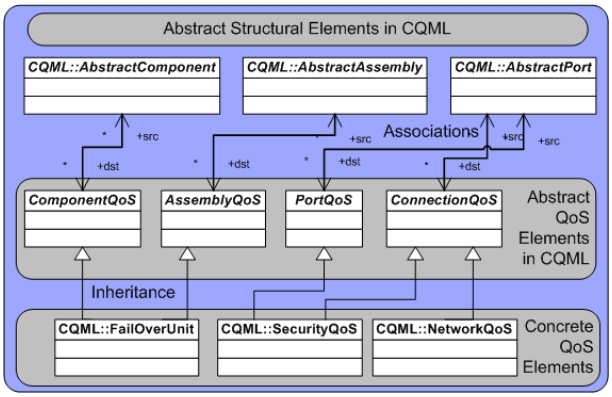

Figure 8: Simplified UML Class Diagram of the Meta-model of CQML.

A very compelling analogy from the programming language world for these abstract types in CQML is an interface in Java or an abstract class in C++. The concrete structural elements in the base language inherit from the abstract structural types. By virtue of inheritance, all the roles and associations in which the abstract base entities can participate in, the derived concrete types can also participate. As governed by the principle of substitutability, the modularized QoS properties can then be associated with derived types as well.

The Figure 7 and the Figure 8 together show how QoS modeling capability of CQML can be superimposed on a base structural language using meta level language composition and simple inheritance mechanism. Using the join point model of CQML and language composition, we address the challenge described in Section 2.1

3.2 Resolution 2: Expressive Modularization of QoS Concerns

Based on the minimal and optional characterization of the underlying component composition language, CQML builds an extensible QoS aspect modeling layer over it. CQML has an ability to bind declarative QoS aspect to one or more of the invariant properties of the underlying base language. CQML categorizes them into five basic abstract types: *Component-QoS*, *Connection-QoS*, *Port-QoS*, *Assembly-QoS* and *Method-QoS* as shown in Figure 8.

The above five abstract types constitute the generic QoS aspect modeling framework in CQML. As the name suggests each abstract

QoS is associated with its corresponding building block in the composition language if available. CQML also allows a particular QoS to participate in more than one category. In the following, we describe each type of category in detail and show how different concrete QoS aspects can simultaneously belong to more than one of the categories.

3.2.1 Extensible Design of CQML

CQML can be extended with new concrete declarative QoS aspect modeling capabilities by inheriting from a basic set of abstract QoS types. To enhance CQML with a concrete QoS characteristic, a language designer has to enhance the metamodel of CQML at one or more well-defined points of extension represented by the four basic abstract QoS types. The concrete QoS elements simply derive from the abstract QoS elements defined in CQML depending upon the category to which they belong. A language designer who wants to add a new type of declarative QoS aspect to CQML has to decide the category to which the new QoS aspect belongs. By doing so the concrete modeling entities inherit the abstract syntax, static semantics, relationships, and integrity constraints of the abstract QoS entities defined in the meta-model of CQML.

These entities constitute the generic QoS aspect 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.

3.2.2 QoS Aspect Modeling Extensions in CQML

We now describe how CQML enables declarative QoS aspect modeling for the five invariant building blocks provided by the underlying base language using our extensible design. Modelers create system models with declarative QoS aspects associated with components, ports, connections, assemblies, or methods as indicated in Step 3 in Figure 5. The remainder of this section describes the details of the CQML concrete QoS aspect modeling capabilities shown in Figure 9, which shows an example of three different declarative QoS aspects associated to components. The FailOverUnit is an availability aspect, PortSecurityQoS modularizes security aspect and NetworkQoS modularizes network level QoS aspects. In Section 4.3 we show how the FailOverUnit declarative aspect can be used to weave in availability related concerns in the model using generative capabilities of CQML tool-suite.

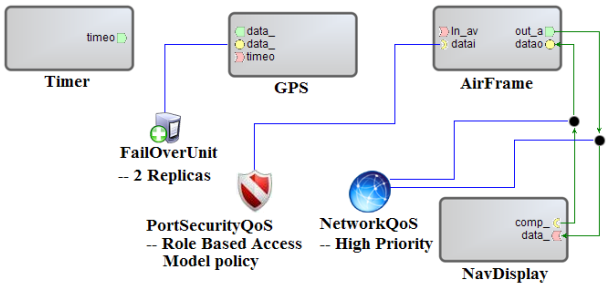

Figure 9: Declarative QoS Aspect Modeling Capability of CQML in Avionics Mission Computing Scenario

Component-QoS Aspect Modeling. In component-based systems, service providers often advertise their functionality with a service level agreement that describes additional guarantees provided by the service in terms of some concrete QoS characteristics. For example, the `GPS` component from Figure 9 has availability requirements and therefore has a FailOverUnit declarative aspect attached to it. CQML allows a modeler to capture the availability aspect of the system through its concrete notation called "Fail Over Unit" (FOU) described next.

A FailOverUnit (FOU) is used to capture the availability concern of one or more components. A FOU is a concrete component-QoS that enables control over the granularity of protected system components, such as a single component or a collection of components. A FOU captures different fault-tolerance attributes, such as the degree of replication of a component and heart beat interval. Based on the availability requirements captured in a FOU, the deployment planning tool described in Section 4 computes a placement for components that satisfies the requirements.

Without a first class notion of a FOU, the availability concern gets tangled with other dimensions of system development. For example, an alternative to FOU would be to model the replicas of the components in the structural view of the system. This is an example of tangling of QoS dimension with the structural view of the system. FOU modularizes the availability concern, and prevents its tangling with the structural dimension.

Connection-QoS Aspect Modeling. Components communicate with each other using logical connections which enable the modeling of system workflows. Quite often, connections themselves have some QoS aspects. CQML allows connection QoS aspects to be modeled. `NetQoS` and SecurityQoS are concrete examples of connection-related QoS concerns in CQML.

The NetQoS element captures network level bidirectional bandwidth requirements of remote procedure calls. Moreover, it categorizes the network traffic represented by connections in disparate traffic classes such as Multi-Media (MM), High-Reliability (HR), High-Priority (HP), and Best-Effort (BE). Such connection level information can be leveraged in component-based systems to preallocate network resources between the physical hosts of the components to provide network level QoS. NetQoS is an extension to the generic connection QoS modeling capabilities of CQML derived from the abstract *Connection-QoS*. This shows that the *Connection-QoS* abstract element provides a join point to bind declarative QoS aspect to connections.

Port-QoS Aspect Modeling. A Port allows components to expose their functionality to other components and provides an end-point for connections between components. Therefore, CQML allows ports of a component to have QoS advice bound to them. Several different QoS concerns can be associated with a port. An example of a connection-QoS in CQML is SecurityQoS aspect. For example, Figure 9 shows a Role Based Access Control (RBAC) model that modularizes security related access control policies. The details of the RBAC security QoS aspect model are beyond the scope of this paper. SecurityQoS is a concrete example of an extension to the generic port related aspect modeling capabilities of CQML. Similar to the *Component-QoS* and *Connection-QoS*, the *Port-QoS* abstract element provides an extension point for potentially many port related QoS.

Component Assembly-QoS Aspect Modeling. Component assembly allows aggregation of one or more components and assemblies. Component assemblies enable hierarchical structuring of the component-based system. Certain types of QoS that we have shown associated with a monolithic component can also be associated with an assembly. For example, an availability requirement aspect can be associated with a component assembly rendering entire assembly as a protected unit of functionality. As mentioned earlier, FailOverUnit modularizes availability concern and avoids

tangling of availability QoS concern with structural decomposition concern. Without a FOU, entire assembly with all the contained components and assemblies will need to be replicated thereby polluting the structural dimension of the system. A FailOverUnit can thus be a *Component-QoS* as well as *Assembly-QoS*. In similar fashion multiple other component assembly QoS characteristics can be defined in CQML with ease by leveraging the extension points provided for a language designer.

Method-level QoS Aspect modeling. Component-based systems often require a capability to model QoS or contract on the interfaces or methods that are implemented by the components. CQML has an abstract notion of a method and the abstract QoS associated with it called *Method-QoS*. If the underlying base language has a first class support for modeling interfaces and/or methods, the abstract elements in CQML can be used to inherit the Method-QoS related associations from it.

In summary, CQML provides concrete graphical syntax to associate declarative QoS aspects to different structural elements of a component-based system at a higher level of abstraction with complete separation from the structural concern. This design of CQML helps us resolve the second challenge of providing an extensible way of expressing QoS design intent in the form of declarative QoS aspects.

3.2.3 Visualizing QoS Aspects

Visualization of QoS aspect models in CQML is quite flexible. System developers and designers often find it intuitive to visualize the QoS characteristic annotations overlaid on top of a duplicate view of the system structure. This is because often the functionality of the system is the dominant dimension of decomposition and developers are often trained and skilled in manipulating the functional aspect of the system. Overlaying QoS on top of structure helps improve comprehensibility of the system as a whole along with its secondary QoS concerns. Such a feature does not violate the principle of separation of crosscutting concerns because the structural view of the system with QoS concerns superimposed is not the primary view of manipulating system structure. While modeling the modularized QoS concerns using CQML, the structure of the underlying system is an optional feature and is not strictly necessary.

3.3 Resolution 3: Automated Generation and Weaving of QoS Advice

We leverage the join point model of CQML and the ability to associate declarative QoS aspects to the structural elements of the system to conduct platform-independent analyses of the structural properties of the system. As mentioned in Section 2.3 several different analyses based on structural property analyses such as component collocation optimization [1], properties of orchestration of component workflow such as capturing end-to-end deadline, annotating component path segments for monitoring, and deployment planning [26]. The analysis phase is represented by Step 4 in Figure 5. In the evaluation section we show how we have used our deployment planning tool that takes into account availability requirements modeled using CQML to generate a placement for components that meet the availability requirements.

In MDE-based system development methodology, the model of the system is the primary artifact for designing and reasoning about it. Moreover, platform-specific metadata is created based on models using the embedded tool support. In order to generate metadata such as deployment descriptors for a specific platform, platform-specific model must be populated. Therefore, any analysis tool that produces results that can be represented by the modeling tool must go back into the model.

For example, our deployment planning tool that gives a placement of components on physical hosts should be viewable in the deployment model of the underlying base language. It allows seamless continuation of the system development life-cycle. To achieve this the deployment model of the underlying base language should be structurally compatible with that of CQML's. Adherence to a common structure allows the CQML's model weaver to push the deployment planning results back into the model in a generic fashion that is independent of the underlying base language.

We use the Constraint-Specification Aspect Weaver [6] (C-SAW) and Embedded Constraints Language (ECL) to populate the model with the result of the analysis tool. ECL allows modularization of commonly required steps while modeling a particular aspect. For example, in our case, populating the results of deployment planning tool is an aspect of modeling that can be modularized using ECL. Aspects are further divided into strategies that can be selectively applied on the models that meet some predicate. We have developed generative capabilities to completely automate the process of generating ECL code for weaving the deployment aspect into the models. This eliminated the learning curve of the modeler to understand how ECL works and simplifies his/her job.

3.3.1 Generative Capabilities of CQML Tool-suite

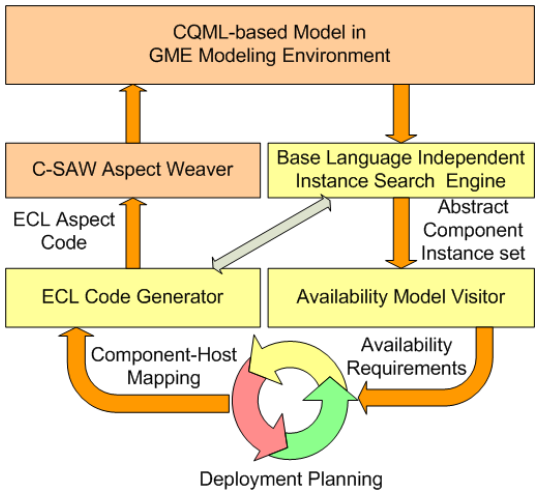

Figure 10: Architecture of CQML ECL Code Generator

This section describes in detail the capability of CQML tool-suite to automatically weave models from descriptive QoS aspects as indicated in the last step in Figure 5. Figure 10 shows the overall architecture of the our generative tool called: *ECL Code Generator*. ECL Code Generator is divided in three main parts: Instance Search Engine, Deployment Planner, and the Code Generator.

Instance Search Engine It searches and collects the instances of the important structural elements such as components, assemblies, ports in a model. These structural elements are instances of the types defined in the meta-model of the composite language. The Instance Search Engine depends on the fact that the elements that it collects are instances of the types that specialize the abstract elements defined in CQML. For example, in J2EEML′ *SessionBean* should be a specialization of the *AbstractComponent* notion defined in CQML. The output of Instance Search Engine is a set of instances of concrete components. The *Instance Search Engine* filters out the platform-specific type information from the collected components before passing them to the next stage of deployment planning. The deployment planner has no knowledge of whether the set of components is a set of LwCCM components or EJB beans (session beans, entity beans) or any other type of platform-specific component.

Deployment Planner The deployment planner visits the availability models that are attached as declarative QoS aspects to the set of components. Based on the component replication degree and the placement metric [26], the planner generates placement for every component and their replicas if any. The output of the planning stage is a simple component to physical host mapping. The planner is designed to be extensible so that more planning algorithms can be used as different strategies to plug-in different planning algorithms.

Code Generator The code generator takes the component to physical host mapping as an input and generates ECL code, which is a modularized way of capturing manual actions required to do deployment planning and modeling. An example of generated ECL code and the BasicSP model in PICML' is shown in Figure 11.

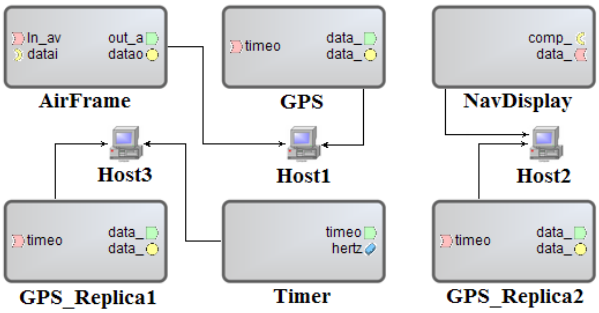

Figure 11: Generated ECL Code and PICML′ Deployment Plan After Weaving

```
defines Deploy, Placement, Association;
strategy Association (dp,host,comp:model) {
   dp.addConnection("Placement",comp,host);
}
strategy Placement () {
   declare dp : model;
   dp := rootFolder().findModel ("DP");
   Association(dp, dp.findModel ("Host1"),
     dp.addModel("Component","AirFrame"));
   Association(dp, dp.findModel ("Host2"),
     dp.addModel("Component","GPS_Replica2"));
   Association(dp, dp.findModel ("Host2"),
     dp.addModel("Component","NavDisplay"));
   Association(dp, dp.findModel ("Host1"),
     dp.addModel("Component","GPS"));
   Association dp, dp.findModel ("Host3"),
     dp.addModel("Component","GPS_Replica1"));
   Association(dp, dp.findModel ("Host3"),
     dp.addModel("Component","Timer"));
}
aspect Deploy () {
   Placement();
}
```

Figure 12: Sample Generated ECL Code

ECL and C-SAW have been used [2] to modularize and automatically weave the deployment planning concern for platform-specific base languages. A drawback of this approach is that the user of C-SAW still has to learn ECL and write ECL code in terms of aspects and strategies to modularize the deployment modeling concern. Our ECL generator completely eliminates the step of writing ECL code and simplifies weaving by automatically generating ECL that does all the steps necessary to populate a deployment plan model of CQML. Moreover, ECL generator works across different different platform-specific composition languages. To enable this it is necessary to decorate the generated ECL statements with platform-specific type information. Figure 12 shows a sample of generated ECL code from the component to host mappings generated by the *deployment planner*. Execution of the ECL code shown above using C-SAW results in the deployment shown in Figure 11. Note that type information is is filtered by the *Instance Search Engine* before deployment planning is done. The ECL Code Generator decorates the generated ECL code with the actual concrete platform-specific type of the components although deployment analysis is performed without platform-specific details of component types. The ECL generator retrieves this information from the Instance Search Engine for each abstract component. The generated ECL is then processed by the C-SAW aspect weaver to populate the model.

This shows that the ECL generator is a powerful aspect code generator for C-SAW that not only modularizes the deployment planning concern but also automates it.

4. Evaluating CQML

This section describes our evaluation of CQML. First we determine the effort required to compose CQML on an underlying base modeling language comparing it with effort required to realize these capabilities directly in the base language. Second, we demonstrate two platform-independent analysis capabilities within CQML showing the automated weaving of analysis results.

4.1 Composability of CQML with Structural Modeling Languages

To evaluate our approach we chose three component-based structural (de)composition languages: the Platform Independent Component Modeling Language (PICML) [1] for Light-weight CORBA Component Model [18] (CCM), J2EEML [27] for Enterprise Java Beans, and the Embedded Systems Modeling Language (ESML) [11] for embedded systems. The feature set of these languages varies greatly – PICML being the most feature-rich language among the three.

There are many commonalities and differences among these languages that stem from the differences in the underlying component model that they model. All of them are component-based system modeling languages and hence treat a component and assemblies (nesting of components and assemblies) as first class entities. For example, J2EEML and PICML support hierarchical composition of assemblies but ESML has a flat, single level notion of an assembly. 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. The notion of ports is present in all the three languages. In J2EEML the ports manifest themselves as invocable beans in an assembly of beans. Table 1 summarizes the similarities and differences between the three languages. In summary, PICML turns out to be an umbrella modeling language that has all the capabilities of the other two.

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 specialized join point model of the three composite languages varies because of the varying structural capabilities of the three base languages. 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 a componentŠs ability 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
Hierarchical Assemblies	Yes	Yes	No

Table 1: Comparison of Capabilities of Selected Three Modeling Languages

Figure 13 and Figure 14 show the models of the avionics mission computing system created using the composite languages. For example, in J2EEML′ QoS advice cannot be associated with a required interface port because there is no support for ports built in to the base language. Similarly, in ESML, declarative QoS aspect cannot be associated with nested assemblies because nesting of assemblies is not supported. The results indicate that CQML can flexibly be composed with the base component-based structural composition languages to provide separation of QoS concerns from the structural concerns.

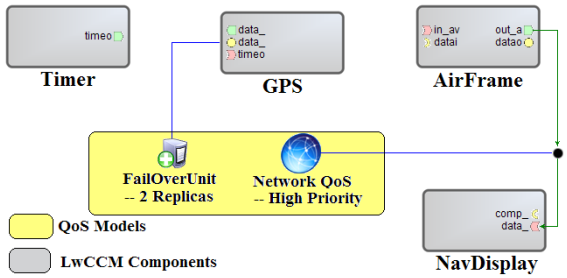

Figure 13: QoS Advice Specification Capabilities of PICML′

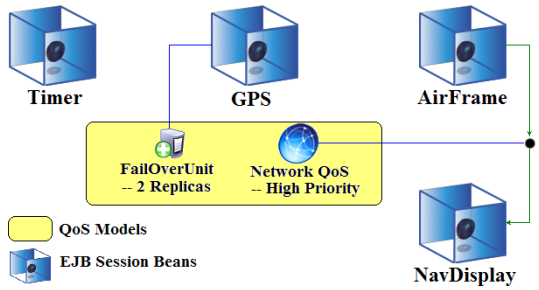

Figure 14: QoS Advice Specification Capabilities of J2EEML′

4.2 Conducting Platform Independent Analysis

In this section we demonstrate how platform-independent QoS-related analysis of structural properties can be conducted using CQML, and how results can be woven back into the base structural language. We focus on the availability analysis.

As described in Section 3, CQML allows us to write platform independent analysis tools without requiring model transformations to suit the format expected by the analysis tool. We developed a variant of our MDDPro [26] deployment planning tool to evaluate the modeling and automation capabilities of CQML. Based on the availability concerns that are captured in the application model, the MDDPro deployment planning tool generates replicas of the protected components and runs a planner on them to decide a placement. The planner is based on the Shared Risk Group [26] (SRG) model that allows us to place replicas in a way that minimizes the risk of simultaneous failure of replicated functionality. MDDPro also enables plugging in different replica placement algorithms to improve system availability.

To evaluate the support provided by CQML, we used the GPS-based avionics scenario shown in Figure 4. We modeled this scenario in the three different base languages with identical set of declarative QoS aspects associated with them. We ran a variant of MDDPro deployment planning tool on each model to generate placement for the primary components and the replicas of the `GPS` component. The planning tool generated three different ECL scripts for three different base modeling languages. The generated ECL scripts automate weaving of the results of deployment planning back into the original model thereby eliminating human efforts. We then used the C-SAW aspect weaver as a vehicle to execute the generated ECL scripts and actually populate the deployment plan models of the base language. Figure 11 shows the result of weaving the deployment decisions back into a model of PICML′ language. These results indicate that deployment planning can be done in a platform independent way using CQML and CQML tool-suite can be used to populate base language models with the deployment decisions.

4.3 Evaluating Savings in Modeling Efforts by Using CQML's Automatic Weaver

In this section, we evaluate the capabilities of CQML to generate aspect code in ECL and how we automate the weaving process using C-SAW [6] by automatically generating ECL code from the component to host mapping given by the deployment planning tool. Figure 13 shows a FailOverUnit attached to the GPS component. The FailOverUnit declarative aspect indicates that 2 replicas (replication degree 2) are desired of the GPS component. Our ECL code generator generates the necessary ECL code for C-SAW that describes where new components need to be created and the number of connections between them. Figure 15 shows a PICML′ model after weaving in the availability aspect into the structural view of the system.

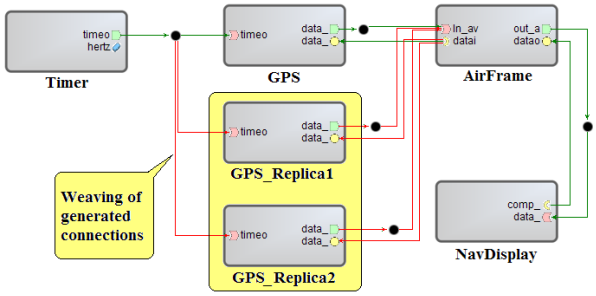

Figure 15: Result of Weaving Generated Components and Connection in PICML′

Table 2 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. The Figure 15 shows only the first case. The number of connections between components grow multiplicatively [26] when replication degree of components increases linearly.

It is clear from the table that without automatic ECL code generation code 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

Replication Degree	Generated ECL LOC	FailOverUnit associated with GPS Component			Generated ECL LOC	FailOverUnit associated with GPS, AirFrame, and NavDisplay Components		
		Component	Connections	Ports		Component	Connections	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 2: Savings in Modeling Effort of Components, Connections, and Ports due to Automatic Generation

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.

5. Related Work

Capturing QoS specifications at design-time has long been a goal of researchers [5, 28]. A prior effort, also called CQML [28], is a platform-independent, general-purpose language for defining QoS properties. CQML [28] allows both interface annotation as well as component type annotation. CQML [28] has support for UML integration based on a lightweight QoS profile and has QoS negotiation capabilities. CQML [28] also supercedes almost all the previous work on QoS specification languages including QML [5] (QoS Modeling Language) and QuO [29] (Quality Objects).

Therefore, we limit our comparison of QoS specification languages to the CQML developed by Aagadel. 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 CQML. 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 Aadedal's CQML, instance level QoS specification is possible in our CQML. Tool support for Aagadel's CQML does not investigate ways of providing separation of QoS concerns throughout other stages of the system development lifecycle, such as deployment planning and configuration. In Section 4 we showed that our CQML can successfully separate QoS concerns even in the later stages of system life-cycle by means of automatic weaving of deployment planning decisions into the base models.

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 [17] 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 [19]. 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 he metaprogramming environment can be leveraged in the process. CQML has not been developed as lightweight profile for UML because UML being a general purpose modeling language, it lacks a component model, which is absolutely essential for successful reuse of CQML. In order to reuse CQML, a richer and more domain (component platform)-specific composition modeling language is necessary.

The SysWeaver [3] 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 perfrom 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.

Another approach [4] 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 [9] 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.

CEA-Frame [16] integrates MDE and AOM techniques to model application variants in platform-independent terms and to automatically transform PIMs to PSMs. The alternative application variants are deployed using platform independent specifications, called service plans. The service plans are known as deployment plans in CQML. CQML takes a more platform independent approach without relying on PIM to PSM transformations and also automates the weaving of the results of the deployment planning back into the original models.

Finally, the model-level aspect weaving approach adopted by CQML is similar to model transformations [10] based on graph transformations [21]. In general, model weaving is considered to be a special case of model transformation. Although model transformation is more general, applying it to analysis such as deployment planning makes transformation rules extremely complex. A procedural approach adopted by CQML is much more suitable for this reason.

6. Discussion

This section discusses our thoughts on CQML and where it fits within the scope of AOSD. Additionally it summarizes its benefits and limitations along with our plans for further improvement.

6.1 CQML in the AOSD Space

Separation of crosscutting concerns and the ability to weave these concerns based on a join point model are the key concepts in AOSD. CQML supports these key concepts by defining a join point model based on which multiple para-functional crosscutting concerns, notably QoS, of component-based systems can be modularized. The specification of QoS is an integral artifact of any system design process. The ability to modularize QoS concerns and reason about their impact on the final characteristics of the system to be deployed has always been an interesting research problem since QoS is one of the many para-functional properties that crosscut the primary design concern of applications. To that end, we have demonstrated how CQML allows different kinds of QoS-specific aspects to be represented.

CQML also identifies the need to weave back the results of the analysis (Step 4 of Figure 5) process into the original system design so that any platform-specific tools associated with the base framework can leverage these results seamlessly. This weaving back of the results is done using the same join point model used for modularizing and associating the para-functional concerns.

Since CQML operates at the level of MDE frameworks, it lies in the category of aspect-oriented modeling (AOM) tools. CQML cannot be considered as a general-purpose aspect language (GPAL) since unlike GPALs like AspectJ, it deals primarily with para-functional properties of component-based systems. At the same time it is applicable to a range of structural modeling languages that satisfy a small set of invariant properties. CQML is similar in concept to a domain-specific aspect language (DSAL) [15] though certain properties of DSALs, such as a language compiler, do not apply to CQML.

6.2 Benefits of CQML

Following are the benefits of using CQML with MDE tools for large-scale component-based system design.

a. *Reuse:* CQML provides a platform-independent approach to modularize QoS concerns during system design that captures well-known patterns and practices in QoS specification. This eliminates the need to reinvent the wheel for the modelers of platform-specific languages, which results in a substantial savings in effort and improved productivity as shown in Table 2.

b. *Extensibility:* Designers of the base language can add their own declarative QoS aspect models and extend the existing set of declarative QoS aspect modeling support in CQML in a seamless fashion because of its extensible design.

c. *Modularity:* CQML modularizes the crosscutting QoS concerns into platform-independent declarative aspects which enhances reuse, extensibility and automation while also giving rise to a model-level DSAL [15].

d. *Generative capability:* CQML generates model weaving code in ECL for the deployment aspect, which can otherwise be very tedious and error-prone to write and validate manually. It also eliminates the learning curve to use AOM weavers, such as C-SAW.

e. *Leverage platform-specific model interpreters:* The artifacts automatically woven back into the system models can be seamlessly used by existing model interpreters associated with the base modeling language. For example, if deployment information or additional replicas are woven back into the original system composition model, an existing deployment plan generator can use this additional information to synthesize the descriptor metadata that is understandable by the component platform.

f. *Restricted set of analysis:* In its current form CQML provides first-class platform independent notions of the system structural building blocks, which lends well for performing structural analyses, such as deployment planning (as demonstrated in Section 4), simple schedulability analysis such as rate monotonic scheduling, footprint optimizations such as component fusion [1] to minimize footprint and improve performance, and security policy domain based system partitioning.

g. *Extensible to other para-functional concerns:* Although CQML deals primarily with QoS issues, our framework is general enough to be extended to other para-functional properties, such as the configuration concern.

6.3 Limitations of CQML and Future Work

a. *Lack of sophisticated analyses:* Complex analyses such as real-time schedulability, reliability, and stability analyses cannot be performed using the current features of CQML. These analyses are dependent on overall system behavior and the behavior of its individual components, and therefore use other formal analysis domain models such as I/O automata, Stateflow, petri nets among others. The structural abstraction support in CQML is generally inadequate to perform analyses based on the above mentioned formal models in a platform-independent manner. We are investigating how the metamodel composition techniques can be used to add behavior and QoS to the structural models. We are also working on determining what kind of and how much data can be extracted using just the existing join point model and metamodel reflection. This will enable us to determine how we can use this data to feed external analysis tools (*e.g.*, AIRES [8], VEST [24]).

b. *Simultaneous multi-QoS management:* CQML at the moment lacks support for analyzing the tradeoffs between multiple QoS dimensions, such as fault tolerance, security, and timeliness. However we do demonstrate CQML's capabilities by analyzing the impact of availability provisioning on the deployment planning aspect of the system. Our current work-in-progress includes analyzing the impact of replication on real-time schedulability and analyzing the impact of replication of confidential data and components on the security of the overall system.

7. Concluding Remarks

Large-scale component-based systems often incur secondary para-functional concerns comprising quality of service (QoS), and deployment planning, which crosscut the primary concern of system functional composition. The scattering and tangling of these secondary concerns impede comprehensibility, reusability and evolution of component-based systems. A Model-Driven Engineering (MDE)-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 MDE-level modularizing abstractions and join point models.

This paper provided three key contributions to address the challenges in MDE tools for component-based system development. First, it described a reusable, platform-independent Component QoS Modeling Language (CQML) that defines a common join point model for component-based system modeling languages to attach declarative QoS aspects. CQML allows separation of crosscutting QoS concern from the functional composition concern of the system. We showed how availability and security policy modeling concerns are modularized in CQML using FailOverUnit and SecurityQoS. Second, we showed that CQML allows the developers to design and develop QoS-based structural analysis tools once and apply them to multiple platforms-specific composition

modeling languages. This obviates the need for transformations of platform-specific models to target analysis domain models to perform QoS analysis. Finally, we demonstrated automatic weaving of analysis decisions generated by a deployment planning tool back into the platform-specific models.

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

References

[1] 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.

[2] Krishnakumar Balasubramanian, Aniruddha S. Gokhale, Yuehua Lin, Jing Zhang, and Jeff Gray. Weaving deployment aspects into domain-specific models. *International Journal of Software Engineering and Knowledge Engineering*, 16(3):403–424, 2006.

[3] 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.

[4] 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.

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

[6] Jeff Gray, Ted Bapty, Sandeep Neema, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan. An approach for supporting aspect-oriented domain modeling. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*, pages 151–168, Erfurt, Germany, September 2003.

[7] Jeffrey Gray, Ted Bapty, and Sandeep Neema. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, pages 87–93, October 2001.

[8] Zonghua Gu, Sharath Kodase, Shige Wang, and Kang G. Shin. A Model-Based Approach to System-Level Dependency and Real-time Analysis of Embedded Software. In *Proceedings of the IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'03)*, pages 78–85, Washington, DC, May 2003. IEEE.

[9] 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.

[10] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003. `http://www.jucs.org/jucs_9_11/on_the_use_of`.

[11] 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 Conf.*, Los Alamitos, CA, August 2002. IEEE Computer Society.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[13] 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.

[14] Ákos Lédeczi, Greg Nordstrom, Gabor Karsai, Peter Volgyesi, and Miklos Maroti. On Metamodel Composition. In *Proceedings of the 2001 IEEE International Conference on Control Applications (CCA)*, pages 756–760, Mexico City, Mexico, 2001. IEEE.

[15] C. Lopes, R. Filman, T. Elrad, M. Aksit, and S. Clarke. *Aspect-Oriented Programming: A Historical Perspective. In Aspect-Oriented Software Development*. Addison Wesley, 2004.

[16] Sten Lundesgaard, Arnor Solberg, Jon Oldevik, Robert France, Jan Øyvind Aagedal, and Frank Eliassen. *Distributed Applications and Interoperable Systems (J. Indulska and K. Raymond Eds.)*, chapter Construction and Execution of Adaptable Applications Using an Aspect-Oriented and Model Driven Approach. Springer LNCS, Berlin / Heidelberg, 2007.

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

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

[19] 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.

[20] P. Tarr and H. Ossher and W. Harrison and S.M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering*, pages 107–119, May 1999.

[21] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific Publishing Company, jan 1997.

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

[23] John A. Stankovic, Prashant Nagaraddi, Zhendong Yu, Zhimin He, and Brian Ellis. Exploiting Prescriptive Aspects: A Design time Capability. In *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded Software*, pages 165–174, New York, NY, USA, 2004. ACM Press.

[24] John A. Stankovic, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis. Vest: An aspect-based composition tool for real-time systems. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 58, Washington, DC, USA, 2003. IEEE Computer Society.

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

[26] 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)*, Durham, New Hampshire, USA, 2007.

[27] Jules White, Douglas C. Schmidt, and Aniruddha Gokhale. Simplifying autonomic enterprise java bean applications via model-driven development: a case study. *Journal of Software and System Modeling*, 2007.

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

[29] 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.