A Platform-Independent Component QoS Modeling Language for Distributed Real-time and Embedded Systems

Sumant Tambe*[1], Akshay Dabholkar[1], Amogh Kavimandan[1], Aniruddha Gokhale[1], and Sherif Abdelwahed[1]

Vanderbilt University, Department of EECS, Nashville, TN
{sutambe,aky,amoghk,gokhale,sherif}@dre.vanderbilt.edu

Abstract. Distributed Real-time and embedded (DRE) systems require multiple, simultaneous quality of service (QoS) properties, such as predictability, reliability and security for their correct operation. With increasing focus on composing DRE systems from components, it becomes necessary for system designers to ensure that the system composition and its QoS configurations are functionally and systemically compatible. Different dimensions of QoS, however, tend to conflict with each other requiring design-time QoS tradeoff analysis. This paper describes a model-driven engineering (MDE) approach to model and analyze the validity of the DRE system QoS properties. First, we describe the Component QoS Modeling Language (CQML), which enables visual separation of concerns in QoS modeling while internally maintaining a unified view of the system. Second, we illustrate how well-defined formalisms of system behavior defined in CQML can be leveraged to conduct QoS tradeoff analysis, such as Real-time schedulability analysis which is described.
Keywords: MDE, DSMLs, QoS modeling and analysis, Fault-Tolerance, Real-Time, Security, Schedulability.

1 Introduction

As the software architectures for large scale, distributed Real-time and embedded (DRE) systems found in domains, such as avionics mission computing, medical systems or electric power grid, move away from traditional stovepiped, closed architectures to more composable, open architectures, system developers are often faced with the daunting challenge of provisioning and validating the different quality of service (QoS) properties of the system. These challenges arise since multiple QoS properties, such as predictable latencies, reliability, availability and security, often mutually conflict with each other in unpredictable ways making it hard to analyze the system for correctness. Without a proper formal tool support, however, DRE system developers have to depend on *ad hoc* techniques to provision and validate the system's QoS requirements, which leads to long iterative development cycles that are both expensive and void of a mathematical proof of system correctness.

* Contact author

This research was supported in part by a Raytheon IRAD grant

The traditional stovepiped approaches to system development, although inflexible and inextensible, have successfully demonstrated the use of formal tools and techniques to verify and validate different system properties although on a smaller scale and for well defined, closed architectures. It is desirable therefore for research and development in transitioning and scaling these formal tools and techniques to the new style of developing DRE systems based on composition. Model-driven Engineering (MDE) [**?**] provides a promising solution to realize these objectives since it provides scalable and intuitive abstractions to system development while also decoupling different stages of system development from each other, such as modeling of the system from the analysis of the system properties.

This paper describes in detail the Component Quality Modeling Language (CQML) MDE framework, which comprises the CQML domain-specific modeling language (DSML) [**?**,**?**] and a set of generative programming mechanisms [**?**] that enables the seamless integration of system analysis tools. CQML assumes that the DRE systems of interest use component-based software development processes including the use of component middleware technologies like CORBA Component Model or Enterprise Java Beans. In the context of our CQML MDE solution, therefore, we leverage our earlier R&D on the Platform Independent Component Modeling Language (PICML) [**?**] to model the DRE system compositions using the component assemblies provided by the PICML DSML.

CQML superimposes QoS modeling abstractions on DSMLs, such as PICML, by providing intuitive modeling abstractions to model different DRE system QoS properties including real-time, fault-tolerance and security requirements of the DRE system at different levels of granularity including component level, component port level and component assembly level. The hallmark of CQML is its ability to provide a clean visual separation for modeling different QoS properties yet maintain an underlying unified framework. The unified framework is used to check for QoS conflicts and tradeoffs using built in constraints as well as via rigorous QoS tradeoff analysis using different back end analysis tools.

The remainder of this paper is organized as follows: Section 2 describes a motivating DRE system we use to describe CQML features; Section 3 describes the features of CQML elaborating on the modeling language that provides a visual separation of concerns in QoS modeling while unifying these internally; Section 4 uses an example schedulability analysis technique to demonstrate how to perform QoS tradeoff analysis in CQML; Section 5 describes related research; and Section 6 describes concluding remarks outlining lessons learned and future work.

2 Motivating use of MDE Approach to Specify and Analyze QoS Requirements

This section uses a DRE system case study to elucidate the need for higher levels of abstraction than those provided by third generation programming languages or declarative mechanisms like XML for specifying and analyzing the multiple QoS properties of DRE systems. We focus on a Robot Assembly case study, which we developed in conjunction with colleagues at Lockheed Martin. Using this DRE system's QoS prop-

erties we highlight the desired characteristics of a MDE tool like CQML described in this paper.

2.1 Robot Assembly Case Study

The Robot Assembly consists of a *production assembly line* that is used in the manufacturing of various goods (*e.g.,* wrist watches). A number of hardware and software modules interact with each other in the production cycle. Humans interact with the modules of the assembly in order to (1) specify and/or change a particular watch order, and (2) monitor the production of watches and intervene in case of assembly malfunctioning (*e.g.,* creation of watches that fail product quality tests, owing to mechanical or software faults in one of the assembly modules) or emergency (*e.g.,* fire, intrusion alarm).

Figure 1 shows the overall Robot Assembly structure, and interactions between various (hardware and software) assembly modules. The Robot Assembly software modules are the `ManagementWorkInstruction (MWI)`, the `HumanMachine-Interface (HMI)`, the `PalletConveyerManager(PCM)`, the `RobotManager(RM)`, the `WatchSettingManager(WSM)`, and the `ClockHandler(CH)`. `MWI` and `HMI` components expose interfaces for the hardware units (*e.g.,* radio, order validation switches) such that human assembly operators and/or maintainers can interact with the Robot Assembly. On the other hand, `PCM`, `RM` and `WSM` components interact with pallet moving and assembly line tools and robot arm, respectively.

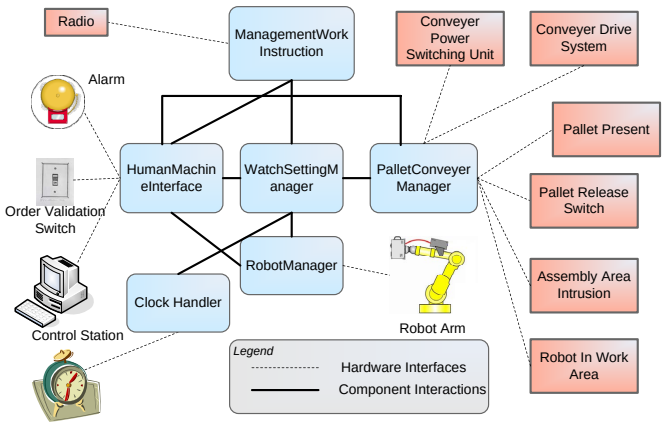

Fig. 1: Robot Assembly Model

A human operator uses the radio unit to specify a work order for a new watch, which is conveyed to the `WSM`. Based on the watch settings the robot control is loaded with the right software. WSM then indicates to the `PCM` to move the pallet to the position, for

which different types of responses can be generated. The WSM then indicates to the RM to process the pallet. The CH component provides accurate timing control. Communication between the components uses events.

This system demonstrates both real-time and fault-tolerance requirements. For example, both the PCM and RM are required to respond to events from WSM in real-time. Similarly the RM software must be made fault tolerant to provide high availability.

2.2 QoS Design Challenges in Robot Assembly

A DRE system, such as the Robot Assembly, can be developed using contemporary component middleware platforms. For example, we have used the Component Integrated ACE ORB (CIAO) [**?**] QoS-enabled component middleware to compose the system from individual components. Although component middleware provides the mechanisms to compose and deploy component-based systems, it does not provide the means to reason about the system correctness nor does it provide intuitive means to specify multiple QoS properties. We elaborate on the QoS design challenges below.

Challenge 1. Expressing QoS design intent. Contemporary QoS-enabled component middleware typically adopt XML as the means to declaratively specify QoS properties for DRE systems. Although such a capability ensures that the DRE system and its QoS design can evolve and change independently, specifying and modifying the QoS specification itself is a tedious and error-prone activity. The key problem with the abstraction level at which the metadata-based QoS specification operates is its lack of expressive power to convey the intent of the QoS designer.

Ideally, the system designers should be able to describe the system's intended QoS characteristics using higher level intuitive, abstractions from which the QoS configuration options pertaining to the underlying component middleware can be automatically generated. Successive changes to the intended system QoS characteristics are done at the higher level of abstraction only. This philosophy is not unlike the concept of *domain workbench* from intentional programming [**?**].

Challenge 2. Modeling crosscutting QoS concerns. As mentioned earlier, DRE systems such as the one shown in the Figure 1 often have simultaneous QoS requirements. For example, *Timeliness*, *Dependability*, *Security* can be considered to be QoS concerns that determine various QoS configuration options for a DRE system developed using component-based technologies.

The multiple different QoS aspects often affect each other in complex and unintuitive ways leading to the configurations of middleware determined by each property to crosscut and conflict with each other. Tradeoffs have to be made in the quality levels of different QoS aspects to satisfy the overall QoS requirements of the system. DRE developers are domain experts who lack a through understanding of such middleware-level crosscutting effects due to different QoS aspects. Without a capability to reason about these crosscutting impacts, the QoS design becomes difficult to develop and and more importantly evolve, as the system evolves.

For example, in our case study, one of the requirements on RM component is that it should have fault-tolerance capability to ensure continuous production on the assembly line. One way to satisfy such a requirement is to replicate the component and synchro-

nize the replicas. However, this extra actions could potentially adversely impact the real-time schedulability of the system violating the system deadlines.

Challenge 3. Need for design-time tradeoff analysis of QoS. The potential solutions to the above two challenges would necessarily create a tension between them because the goals are naturally conflicting. QoS intents are bound to conflict with each other at certain sufficiently high levels of quality. Therefore, a tradeoff analysis is necessary to balance the forces between them. Such a tradeoff analysis requires the high level design intents to be integrated to produce a coherent set of middleware QoS configuration metadata to deploy the system.

If the dependencies of configuration options of different QoS characteristics are not resolved at design time, the potential mis-configurations may go undetected till much later in the system development cycle. Depending upon the DRE system under consideration, identifying the cause of such failures and correcting the failures may be prohibitively expensive or infeasible. Design time analysis of QoS aspect dependencies helps prevent costly and difficult to debug failures at later stages.

3 The CQML MDE Approach for QoS Specification and Analysis

In this section, we describe the design of the Component QoS Modeling Language (CQML), a domain specific modeling language (DSML) that allows DRE system developers to express QoS design intent at different levels of granularity using intuitive visual representations. CQML has been developed using the Generic Modeling Environment (GME) [**?**] toolkit. CQML is platform-independent since its modeling capabilities can be used to specify QoS designs for systems that have been developed using any component-based technologies, such as CORBA Component Model (CCM) and Enterprise Java Beans (EJB).

3.1 Underlying Assumptions for CQML

Our focus is on DRE systems composed of components that are interconnected in the form of workflows. We assume that a component can either be a single indivisible unit of functionality or a collection of components assembled together as a reusable and deployable unit. Since it is aimed specifically at modeling non-functional characteristics (*i.e.,* QoS characteristics) of DRE systems, the design of CQML requires an underlying modeling language that allows the construction and manipulation of component models in a graphical environment. We refer to such underlying component modeling languages as *system composition modeling languages*, which includes the Platform Independent Component Modeling Language (PICML) [**?**] developed by our group in prior research.

An intuitive and natural way of associating QoS characteristics to the components modeled in a system composition language is to overlay the QoS characteristics on the system structure. Such an approach improves comprehension of the system as a whole along with its QoS characteristics. When modeling the QoS characteristics, the structure of the underlying system should be available only as a reference to the QoS modeler so as not to allow modifications to the underlying system structure and topology.

To facilitate modeling and comprehension of the impact of multiple QoS characteristics for component-based DRE systems, we have developed the Component QoS Modeling Language (CQML) that can enhance any component-based system composition modeling language . CQML leverages the system structure modeling capability from the underlying system composition modeling language. CQML therefore requires the underlying language to expose a minimal set of structural capabilities summarized below.

- **Component.** The composition language should treat the concept of a component as a first class entity. A component embodies a reusable unit of functionality (either as a monolithic entity or a reusable assembly) that can be deployed independent of other components in the system.
- **Connection.** The system workflow comprising inter-operating components is captured by connections in component-based systems. The structuring language should therefore treat a connection as a first class entity.
- **Port.** A port is an application-level communication endpoint abstraction exposed by a component to establish one or more connections with other components.
- **Component Assembly.** As noted earlier we require a reusable component assembly to be treated as a first class entity. Having such a feature in the language improves scalability of the models constructed using that language. An assembly component is an important concept since QoS can be associated with a group of components.

3.2 CQML Modeling Capabilities

Based on the above minimal characterization of the underlying component composition language, CQML builds an extensible QoS modeling layer over it. CQML has an ability to associate QoS characteristics to one or more of the foundational building blocks of the underlying system composition language. CQML categorizes them into four basic abstract types: *Component-QoS*, *Connection-QoS*, *Port-QoS*, and *Assembly-QoS*. The above four abstract types constitute the generic QoS modeling framework in CQML. A broad spectrum of QoS aspects that are relevant to the domain of DRE systems fall under one or more of the above categories. 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 that one of them.

Extensible Design of CQML CQML can be extended with new concrete QoS 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 meta-model of CQML at one or more well-defined points of extension represented by 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 concrete QoS element to CQML has to decide the category to which the new QoS model belongs. By doing so the concrete modeling

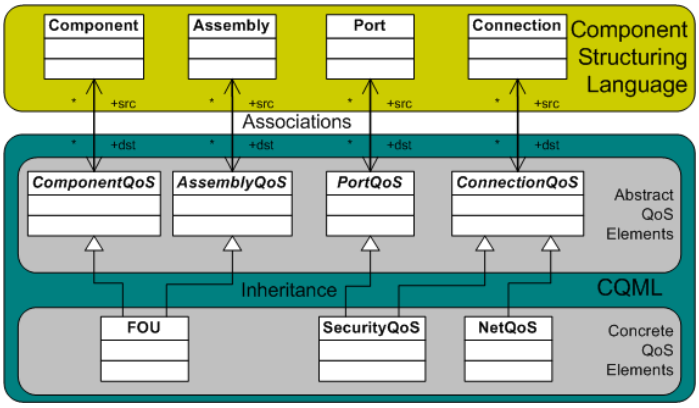

Fig. 2: Simplified UML class diagram of the meta-model of CQML.

entities inherit the abstract syntax, static semantics, relationships, and integrity constraints of the abstract QoS entities defined in the meta-model. These entities constitute the generic QoS modeling framework of CQML. Although designing a new language construct–in this case a new QoS characteristic–is an extremely thoughtful process, a significant portion of design decisions are already taken for the language designer in the generic QoS modeling framework of CQML. The reuse promoted by CQML design and its generic QoS entities thus lends itself to easier DRE systems-specific modeling enhancements. It prevents reinvention of previously designed artifacts for every new QoS characteristic that is added.

QoS Modeling Extensions in CQML We now describe how CQML enables QoS modeling for the four foundational building blocks provided by the underlying system composition modeling language using our extensible design. Figure 3 illustrates how the structural model of the case study described in a earlier section, the Robot Assembly, can be associated with multiple QoS aspect models. The remainder of this section describes the details of the CQML concrete QoS Modeling capabilities.

Component-QoS modeling. In component-based systems, service providers often advertise their functionality with a Service Level Agreement (SLA) that describes additional guarantees provided by the service in terms of some concrete QoS characteristics. For example, the `Robot Manager (RM)` component from our case study shown in Figure 3 has a dependability aspect. CQML allows a modeler to capture the dependability aspect of the system through its concrete notation called Fail Over Unit (FOU) described next.

A `FailOverUnit` (FOU) is used to capture the dependability aspect 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 monolithic or assembly component. A FOU captures different fault-tolerance attributes, such as the degree of replication of a component and heart beat interval, and allows automatic source code genera-

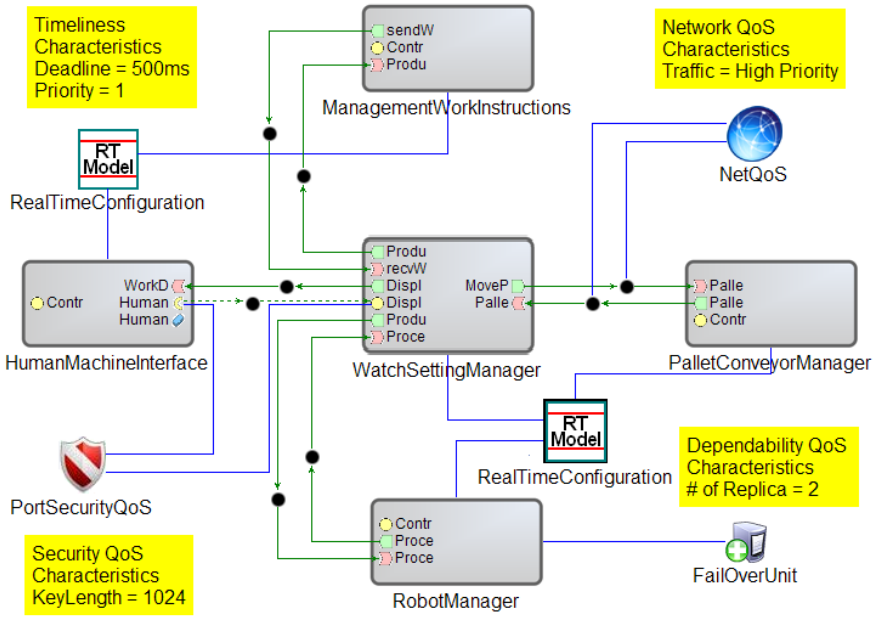

Fig. 3: Multiple QoS aspect representation in the Robot Assembly

tion for the fault monitoring infrastructure. Such infrastructure typically comprises of fault monitors that depend upon a periodic heart-beat beacon and is described in [**?**] in detail.

Likewise, the components in the Robot Assembly guarantee timeliness properties and therefore, they have Real-time configurations associated with them. As shown in Figure 3, CQML allows modeling of Real-time configurations of components as another example of a component related QoS. The `RealTimeConfiguration` model in CQML allows modeling of a deadlines for the tasks in a component as well as relative priorities among them. This information is further used in the QoS analysis, such as schedulability analysis of the system. FOU and RealTimeConfiguration are concrete examples of extensions to the generic component QoS modeling capabilities of CQML. Component-QoS abstract element in the meta-model of CQML provides an extension point for potentially many component related QoS such as FOU and Real-time Configuration.

Connection-QoS Modeling. Components communicate with each other using logical connections which enable the modeling of system workflows. Quite often, connections themselves have some QoS aspects beyond simply being logical place-holders for physical data transportation links. CQML allows connection QoS aspects to be modeled. NetQoS and SecurityQoS are concrete examples of connection related QoS in CQML.

The NetQoS element captures network level bidirectional bandwidth requirements of the remote procedure calls. Moveover, 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 the Robot Assembly to guarantee bounded network latencies between the `WSM` and the `PCM` components to provide network level QoS. Connections might as well have security related QoS attributes associated with them. For example, the type of encryption and key length used for the communication between `HMI` and `WSM` components determine quality of protection in our case study. SecurityQoS captures the security aspect of connections.

NetQoS and SecurityQoS are extensions to the generic connection QoS modeling capabilities of CQML. Similar to the *Component-QoS* abstract element, the *Connection-QoS* abstract element provides an extension point for potentially many connection-related QoS.

Port-QoS 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 be annotated with one or more port related QoS aspects. Several different quality of service aspects can be associated with a port. For example, a port can be annotated with SecurityQoS attributes.

Ports often are the points of control for implementing the security concerns in terms of caller credentials. For example, as shown in Figure 3, the Port-SecurityQoS associated with a port of the `WSM` component dictates the necessary security credentials of a human actor, who uses the `HMI` component to invoke the operations. Moreover, quality of protection parameters such as key-length and encryption algorithms provide different strategies of implementing security QoS at port level.

Thus, SecurityQoS is a concrete example of an extension to the generic port QoS and connection QoS modeling capabilities of CQML simultaneously. The abstract syntax and the relationships of multiple abstract QoS elements can be combined together in a single concrete QoS element. 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 Modeling. Component assembly allows aggregation of one or more components and/or 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, a FOU can be associated with a component assembly rendering entire assembly as a protected unit of functionality. Thus, a FOU not only inherits relationships defined for the generic *Component-QoS* but also that of the *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.

In summary, the modeling front-end provided by CQML captures the QoS design intent at a higher level of abstraction, which is in many ways similar to the concept of *domain workbench* mentioned in the Challenge 1. The design of CQML helps us resolve the first challenge: expressing QoS design intent. Extensible QoS modeling capabilities of CQML can be leveraged to develop a suite of generator tools that are helpful in various stages of DRE system development for example, configuration options checking [**?**]

and deployment metadata generation [**?**]. In this paper we show how design time analysis can be performed to establish system's properties at design time in the face multiple simultaneous QoS.

3.3 Reusability of Model Interpretation in CQML

Building a concrete syntax for a domain specific modeling language is only half the story. A language designer has to develop a language parser/interpreter that visits the elements in a well-formed model efficiently. Substantial efforts are necessary to build such a parser manually and more so for every new QoS element added to the QoS modeling language. For CQML, no reinvention of parsing functionality is necessary because all the concrete QoS entities are extensions of one of the four basic abstract QoS elements defined in the language. As the static semantics, relationships and integrity constraints designed in the generic QoS model remain unchanged, irrespective of the number of concrete QoS elements in the language, reuse of the model parsing functionality becomes possible. It should be noted that parsing of individual concrete QoS elements has to be done by the language designer as it cannot be generalized. This leads to substantial savings in production time for language processing tools and the development becomes less error prone.

For the schedulability analysis, we have developed a new model interpreter called RTAnalysis interpreter that generates metadata for a backend schedulability analysis tool. We also have a FaultTolerance interpreter [**?**] that auto-generates component-based fault-tolerance monitoring infrastructure from a CQML model that has dependability requirements modeled in it. We have developed a SecurityQoS interpreter that generates middleware configuration metadata such as security policies, user permissions, secure protocols/methods to be used between components. We combine the capabilities of these individual model interpreters using an integration framework called the EventBus Framework. The details of the framework are described in the next section.

3.4 Unifying QoS Characteristics via CQML's EventBus Framework

Section 3.2 described the design and implementation of the modeling front end of CQML, which provides an extensible modeling environment for QoS modeling by cleanly separating the modeling of different QoS aspects. The QoS unifying feature in CQML is provided by the *EventBus Framework* illustrated in Figure 4 that allows different QoS interpreters to communicate. Although CQML separates the modeling of different QoS aspects, the EventBus Framework plays a central role in *weaving* together the cross-cutting concerns between different QoS aspects.

Our solution approach selects one QoS aspect as a primary dimension, for *e.g.* real-time QoS, and interweaves it with other secondary dimensions such as fault-tolerance and security which in turn may imply ramifications on effectiveness of the primary one. Thus, the framework will enable the RTAnalysis interpreter to respond to the additions injected into the system QoS model by the FaultTolerance interpreter and SecurityQoS interpreter. We now describe how CQML unifies the QoS characteristics and enables QoS tradeoff analysis to deal with their conflicting nature.

Communication between QoS Interpreters. The EventBus framework is an instance of the anonymous publish-subscribe architecture that allows the members of the EventBus, which are the model interpreters, to receive and respond to the events generated by other interpreters. Interpreters that generate events must express their intent to do so to the framework. Similarly, the interpreters interested in receiving events generated by other interpreters also have to register their interest to receive the events.

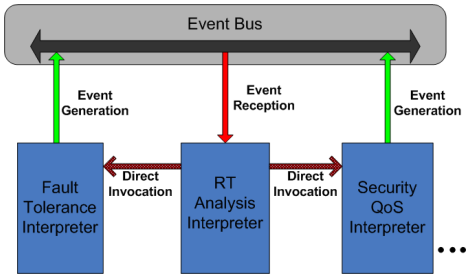

Fig. 4: The architecture of the EventBus framework

Several different categories of events are supported by the EventBus. For example, when the FaultTolerance interpreter and the RTAnalysis interpreter are plugged-in to the EventBus, the FaultTolerance interpreter generates a *periodic computation* event corresponds to the insertion of the collocated HeartBeat (HB) components that generate a periodic heart-beat beacon to enable timely fault detection. This periodic heart-beat beacon is a topic of interest to the RTAnalysis interpreter since this becomes as additional computation and communication task that must be accounted for in system schedulability.

The RTAnalysis interpreter is notified by the EventBus about the *additions* that other interpreters–in this case the FaultTolerance interpreter–are making. The RTAnalysis interpreter in turn responds to the events by querying the FaultTolerance interpreter for the nature of computational behavior that is being added in one or more components. For example, the RTAnalysis interpreter requires the behavior information of the newly inserted functionality to generate schedulability related metadata for a specific back-end analysis tool.

Generality of the EventBus Architecture. The architecture of the EventBus is general enough to support event-based communication between multiple QoS interpreters. For example, one of the side effects of dependability provisioning is the overhead of periodic state synchronization in certain classes of dependable systems. Specifically, our cases study uses semi-active replication for the `RM` component and the replicas are updated with the latest state information from the primary periodically. This state information is critical and may also need to be encrypted by the security policy adopted by the security QoS modeler. Therefore, the FaultTolerance interpreter generates an event representing an *periodic communication* owing to the requirement that periodic state synchronization will happen at runtime between the primary and the replicas.

The SecurityQoS interpreter responds to the *periodic communication* event and incorporates the increase in time required due to the overhead of encryption/decryption of the marshalled state. Finally, it generates *periodic computation* event that corresponds to the periodic, secure state synchronization activity. The RTAnalysis interpreter responds to the *periodic computation* event as it did in the fault-tolerance case and generates nec-

essary information for an analysis tool. Thus the plug-in architecture of the EventBus allows multiple QoS interpreters to participate in QoS trade-off analysis and thus helps resolve the second and the third challenge mentioned earlier in the paper.

4 Analysis Capabilities using CQML

In this section we describe how CQML can be used to analyze the QoS characteristics models and how trade-offs can be made. We use the Robot Assembly case study described in Section 2 to illustrate how CQML weaves in multiple QoS aspects and generates metadata that are used by a back-end real-time schedulability analysis tool. As explained in Section 3.4, we chose the real-time QoS aspect as the primary aspect and weave the effects of other QoS aspects on it for analysis.

We have currently integrated with the Times [**?**] schedulability and model checking functionality through the RTAnalysis interpreter. In order to correctly determine the schedulability of the system it is important to convey the behavioral semantics of the system components. We leverage the behavior modeling capability provided by an input/output automata-based language called Component Behavioral Modeling Language [**?**]. We use it to produce high-fidelity mapping of component behavior into timed automata–the underlying formalism used by the Times tool. Alternatively, the behavior could be fed into the interpreter through other formalisms, such as UML state charts and/or activity diagrams.

The RTAnalysis interpreter obtains the data necessary for schedulability analysis from both: CQML's *RealTimeConfiguration* model and the behavior model of components. It requires the task priorities, execution times, task behavior types (periodic, sporadic or controlled), and deadline to generate input for the Times tool, from which the latter derives worst-case response times (WCRTs). We depend on the modeler to provide the above mentioned information.

Schedulability of Robot Assembly under Multiple QoS Requirements. In order to improve the fault detection rate, an increase in the heart-beat frequency, might have an adverse effect on the schedulability of the critical path. The reason being, with the increase in the fault-monitoring frequency, there is a corresponding increase in the number of instances of the corresponding periodic tasks to be scheduled within the deadline.

The RTAnalysis responds to the *periodic computation* events and produces an updated timed automata of the system behavior as shown in Figure 5. The behavioral semantics map to new process automata containing new tasks and states for the `RM` component. Likewise, while weaving the security QoS aspect in `HMI` and `WSM` components, the additional CPU overhead of encryption/decryption must be incorporated at the port level boundaries of above components. The RTAnalysis interpreter adds an encryption task right before leaving the `HMI` automaton and a decryption task right after entering the `WSM` automaton.

Figure 5 shows a simplified schematic of the timed automata generated by the RTAnalysis interpreter for the Robot Assembly case study that weaves in fault monitoring overhead, state synchronization overhead, and encryption/decryption overhead while performing the schedulability analysis.

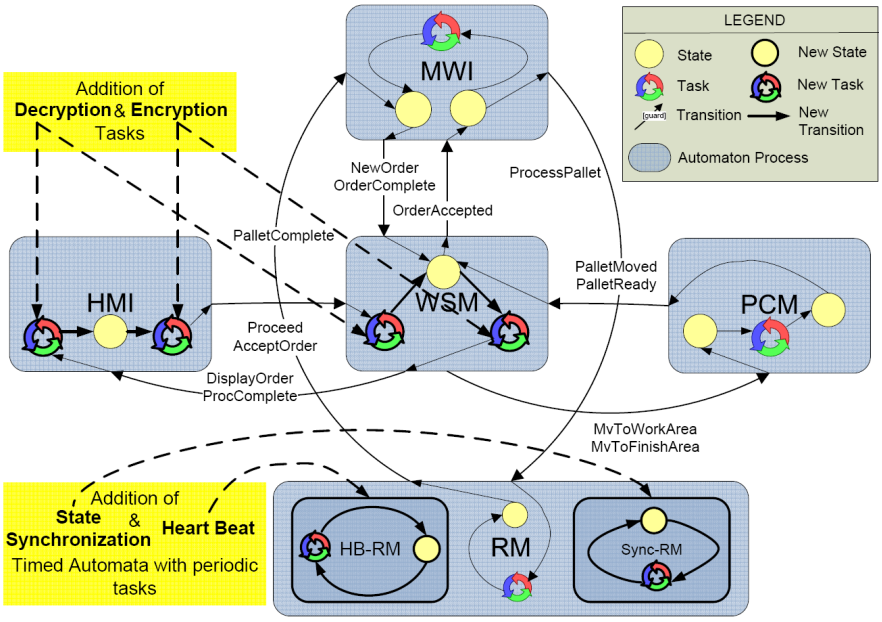

Fig. 5: Effects of multi-QoS interweaving in Robot Assembly

This new model can be analyzed by the Times tool to determine whether the system is still schedulable *i.e.,* meets its real-time deadlines. Thus the interpreter automates the process of determining the effect on system schedulability due to interweaving of other QoS aspects such as fault-tolerance and security. The results from the analysis can be used to fine tune the system's QoS aspect configuration, for example, the heart-beat beacon frequency and/or key length and impart some predictability to the schedulability of the system.

5 Related Work

In this section we discuss the existing work on QoS modeling of component based middleware systems and compare it with CQML and its design-time QoS trade-off analysis capabilities.

The idea of capturing various QoS properties as part of design process of a system is promoted in QML [**?**]. Using QML, developers can define contracts on components at design time. Various QoS options can be clearly separated using QoS categories, which themselves are user defined. QML is used to capture user requirements that are translated into corresponding network and system parameters. A QoS metamodel is described in [**?**] that builds on QML. The authors also discuss in detail the extensions to CCM needed to support the QoS models developed.

AgFlow [**?**] and AMPol-Q [**?**] discuss composition-based QoS specification and middleware adaptation. A quality model is defined in AgFlow middleware platform. It

describes service selection algorithms based on local optimization and global planning, to be used by a composite service application. Being platform-independent, CQML modeling abstractions can be used in conjunction with AgFlow and AMPol-Q. The QoS tradeoff analysis capabilities of CQML can then be directly used by these tools for calculation of exact criteria values during the service selection process.

Q-RAM [**?**] is a QoS-based resource allocation model and a QoS management framework that tries to optimize the resource allocation to satisfy various QoS dimensions such as timeliness, reliability, cryptographic security and other application-specific quality requirements. It is a OS kernel level approach but CQML takes a more higher-level design time approach to unified QoS interweaving and analysis.

CQML differs from the above specification tools in the following ways: (1) These tools are platform-specific, whereas CQML focuses on capturing platform-independent QoS configurations, (2) These tools require modifications to the underlying middleware stack like extending the Interface Definition Language (IDL). In contrast, CQML does not necessitate changes to the underlying middleware itself, but configures the underlying middleware by generating correct platform-specific configurations through model checking, trade-off analysis and interpretation, (3) Though some of the above mentioned techniques define generic quality criteria, they don't allow pluggability of different underlying system structuring languages like CQML does. (4) Moreover, these DSMLs do not provide extensible QoS unification and trade-off analysis capabilities like CQML's EventBus framework and the RTAnalysis interpreter.

6 Concluding Remarks

Distributed Real-time and embedded systems require multiple and simultaneous quality of service (QoS) requirements. Specifying and analyzing the combined effects of these QoS dimensions is a hard problem. This paper described a model-driven engineering (MDE) approach to addressing these challenges. We described the Component QoS Modeling Language (CQML), which is a domain-specific modeling language that provides a visual separation of concerns for QoS modeling, but uses an underlying publish-subscribe mechanism to integrate the different dimensions of QoS. We evaluated the capabilities of CQML using a RobotAssembly case study and showcased the impact of multiple QoS aspects on the schedulability of the overall system.

There are several lessons to be learnt from this work. Separating system's QoS design aspects using higher level abstractions contradicts with the fact that multiple QoS concerns often affect each other in complex ways and therefore, complete isolation from each other is extremely difficult to achieve, if not impossible. Sophisticated tool support and integration capabilities can help raise the bar for modeling languages that capture the QoS design intent. Although model scalability is a selling point of the MDE-based approach like ours, analysis of large, generated formal models is often computationally impractical. Formal model generation technique adopted by our solution should be used judiciously and optimizations should be performed where applicable.