Noname manuscript No.
(will be inserted by the editor)

A Model Transformations-based Approach to Automating Middleware QoS Configurations

Amogh Kavimandan[1*], **Aniruddha Gokhale**[1**], **Anantha Narayanan**[1], **Gabor Karsai**[1]

Dept. of EECS, Vanderbilt University, Nashville, TN 37235.

Received: Feb 1, 2009

Abstract The development and operational life-cycles of distributed real-time and embedded (DRE) systems can be improved using component middleware due to its support for rapid assembly and deployment of large applications. The flexibility of component middleware can, however, complicate DRE system development since non functional system properties, such as system quality of service (QoS), depends on the effective configuration of the middleware. DRE system developers often lack detailed knowledge of the underlying middleware, which makes it challenging for them to map the domain-specific QoS policies into the right set of QoS configurations of the middleware. This paper describes a technique based on model transformations that addresses these challenges. Our technique enables developers to use domain-specific abstractions to specify their QoS policies, which are transformed into middleware QoS configurations using model transformation algorithms we developed. Verifying the correctness of the transformation process itself and that of the generated QoS configurations is addressed using structural correspondence and automated model checking, respectively.

Keywords: Model transformation, QoS policies, middleware configurations, DSMLs.

1 Introduction

Contemporary component middleware technologies, such as Enterprise Java Beans (EJB) and CORBA Component Model (CCM), help to decouple the development of application logic from the provisioning of non functional properties of the system, such as domain-specific quality of service (QoS) policies, which are key to the correct functioning of distributed real-time and embedded (DRE) systems. Examples of DRE system QoS properties that can be configured in the middleware includes different types of concurrency, handling multiple levels of priorities of system tasks, publish/subscribe event-driven communication mechanisms, reliability, security, and multiple scheduling algorithms, among others.

Although component middleware separates the QoS configuration complexity from the application logic, the need to support a variety of domain-specific QoS policies has made the middleware itself very complex, which in turn makes the middleware QoS configuration activity very complex. Middleware QoS configuration thus involves mapping the system-level *QoS policies*—which are dictated by domain requirements—onto the solution space comprising the *QoS mechanisms* for tuning the underlying middleware.

Examples of domain-level QoS policies include (1) the degree of concurrency required to provide a service, (2) the priorities at which the different components should run, (3) the alternate protocols that can be used to request a service, (4) the granularity of sharing among the application components of the underlying resources, such as transport level connections, (5) the number and size of outstanding requests that are permissible at any instant in time, and (6) the maximum and minimum amount of time to wait for completion of requests, among many others. All these domain-specific QoS policies must eventually be mapped onto the right middleware QoS configurations, which includes choosing the right set of middleware configuration parameters and assigning the right values to them.

An additional dimension of complexity stems from the need to determine these QoS configurations at different time scales. This includes *statically*, *e.g.*, directly hard coded into the application or middleware; *semi-statically*, *e.g.*, configured at deployment time using metadata descriptors; and/or *dynamically*, *e.g.*, by modifying QoS configurations at runtime.

The complexity involved in the middleware QoS configuration process raises the following questions:

- How can the domain-specific QoS policies of the system be mapped onto QoS configurations of the underlying middleware, particularly by DRE system developers who

[*] Author has since graduated from Vanderbilt University
[**] Contact Author: a.gokhale@vanderbilt.edu

are domain experts but seldom possess detailed knowledge of the middleware and its configurability?

- How are the right set of configuration parameters determined and how are valid values for the selected set of QoS configuration parameters determined? Are there any *patterns of usage, i.e.*, best practices in configuration that can be used across a variety of application domains?
- How can dependency relationships between configuration parameters be resolved while ensuring that their interactions do not adversely impact QoS? These issues arise at the individual component level (local) as well as at aggregate intermediate levels, such as component assemblies, all the way through the entire application (global), and all of these at different time scales, *i.e.*, design-, deployment- and run-time.

Without scientific techniques and effective tools to address these questions, existing approaches to middleware QoS configurations will often result in QoS mis-configurations that are hard to analyze and debug. As a result, failures will stem from a new class of configuration errors rather than (just) traditional design/implementation errors or resource failures.

Solution approach → Model-to-model Transformations: We address these questions by presenting a technique based on model-to-model transformations, which is realized in the context of a tool-chain called *QUality of service pICKER* (QUICKER) [9]. QUICKER is designed to bridge the gap shown in Figure 1 between:

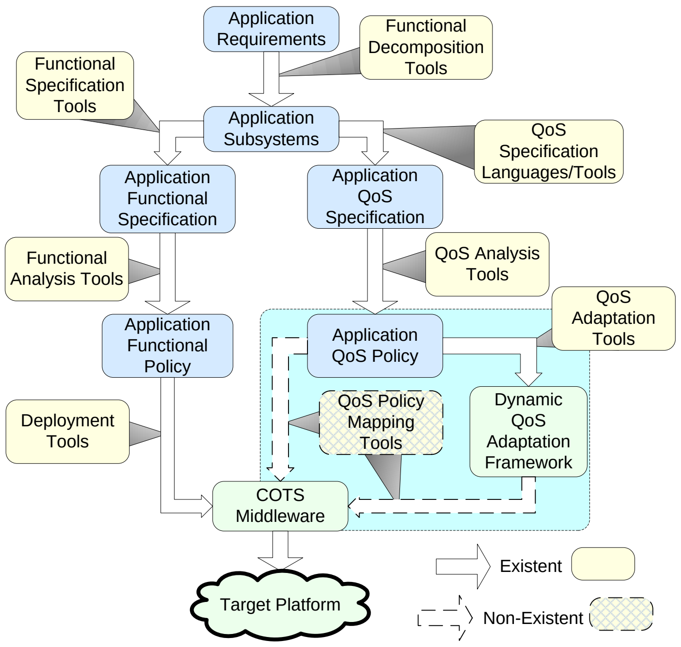
Application Requirements
Functional Decomposition Tools
Functional Specification Tools
Application Subsystems
QoS Specification Languages/Tools
Application Functional Specification
Application QoS Specification
Functional Analysis Tools
QoS Analysis Tools
QoS Adaptation Tools
Application Functional Policy
Application QoS Policy
QoS Policy Mapping Tools
Dynamic QoS Adaptation Framework
Deployment Tools
COTS Middleware
Target Platform
Existent
Non-Existent

Fig. 1: QoS mapping landscape

- **Functional specification and analysis tools**, such as PICML and Cadena [6], that allow specification and analysis of application structure and behavior,
- **Schedulability analysis tools**, such as TIMES [1], AIRES [12], VEST [26], that perform schedulability and timing analysis to determine the exact priorities and time periods for application components, and
- **Dynamic QoS adaptation frameworks**, such as the Resource Adaptation and Control Engine (RACE) [24] and QuO [28], that allocate resources to application components, monitor the QoS of the system continuously, and apply corrective control to modify the QoS configuration of the middleware at runtime.

QUICKER makes the following contributions in addressing the questions raised above:

- QUICKER provides domain-specific modeling languages (DSMLs) [16] that enable DRE system developers to use intuitive abstractions to model system-level QoS policies.
- QUICKER provides a model-to-model transformation process to automate the mapping of domain-specific QoS policies expressed in the DSML into middleware-specific QoS configurations. This transformation process is verified for correctness using structural correspondence [18].
- Dependencies among the configuration options and the correctness of the generated configurations at all time scales is resolved using automated model checking provided by the Bogor [23] model-checking framework.

The remainder of this paper is organized as follows: Section 2 describes the key challenges in QoS configuration of component middleware; Section 3 describes the QUICKER model-to-model transformation approach; Section 4 discusses how we have employed structural correspondence and model-checking techniques to verify the correctness of the QoS configuration process; Section 5 compares QUICKER with related work on model-driven QoS config/-uration/adapta/-tion; and Section 6 presents concluding remarks.

2 Sources of Complexity in Configuring Middleware for DRE Systems

This section describes the challenges in configuring middleware, which is needed to enforce the QoS policies of DRE systems. To better explain these challenges, we describe them in the context of a representative application hosted on a specific middleware – in our case real-time CORBA [21] and the CIAO CORBA component middleware that builds upon it.

2.1 Overview of Real-time CORBA and Component Middleware

Figure 2 illustrates the Real-time CORBA (RTCORBA) middleware architecture. RTCORBA extends traditional CORBA artifacts, such as (a) the object request broker (ORB), which mediates the request handling between clients and servers, (b) the portable object adapter (POA), which manages the lifecycle of CORBA objects, (c) stubs and skeletons, which are generated by an interface definition language (IDL) compiler

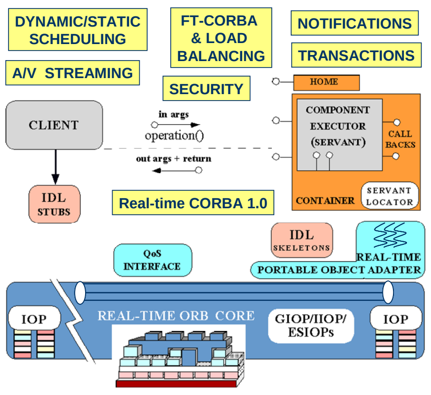

Fig. 2: **Real-time CORBA Middleware**

that hide the distribution aspects from the communicating entities, with real-time policies and interfaces.

RTCORBA defines standard interfaces and QoS policies that allow applications to configure and control (1) *processor resources* via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service, (2) *communication resources* via protocol properties and explicit bindings, and (3) *memory resources* via buffering requests in queues and bounding the size of thread pools.

Some of the important RTCORBA features we use in our examples described in this paper are detailed below. Applications leverage these features by programmatically specifying policies on the POA using standard CORBA operations, such as `create_POA` or `validate_connection`.

- *Thread pools* – RTCORBA supports concurrency and prioritized handling of requests by virtue of thread pools that can be created by enforcing the thread pool policy on the POA that manages the server object. Thread pools may also comprise a special feature called *lanes*. Lanes enable multiple priorities to be assigned to a set of threads so that requests can be executed at different priorities. RTCORBA allows the notion of lane borrowing wherein if lanes of higher priority are exhausted due to multiple client requests, one or more lower priority lanes can have their priorities raised temporarily to handle additional requests. The size of the thread pools can be fixed statically with additional provisions to increase or decrease the size by dynamic creation and removal of threads.
- *Priority bands* – RTCORBA supports a mechanism called priority bands to alleviate priority inversions on network connections due to a lower priority request blocking a higher priority request. Priority bands enable multiple simultaneous connections between communicating peers where each band designates a certain priority of the request.
- *Priority handling* – RTCORBA defines two priority models for handling requests from a client to a server. For requests that have fixed priorities that are determined offline through other tools, the `SERVER_DECLARED` policy is enforced on the POA that handles such client requests.

  If the request handling priority is determined online by a client, this priority is propagated to the server which is made possible by enforcing the `CLIENT_PROPAGATED` policy on the POA that handles these client requests.

The CIAO component middleware is an implementation of the lightweight CORBA Component Model (LwCCM) [20] and leverages RTCORBA. DRE system developers can realize large-scale DRE systems by assembling and deploying CIAO-based LwCCM components. Components within LwCCM can have four different kinds of ports. The cardinality of each port can be one or more. The provided port called a *facet* represents an interface published by the component. The required port called a *receptacle* provides a mechanism to the component to leverage services offered by other components. The other two ports are the *event source* and *event sink* used to publish and subscribe to events, respectively. A component assembly is a reusable block of connected components, which can be viewed as a logical, composite component.

The applications within these DRE systems can use publish/subscribe communication semantics (by using the component event source and sink ports) or request/response communication semantics (by using the facet and receptacle ports). The QoS properties of these applications are realized by appropriately configuring the CIAO middleware and its underlying RTCORBA mechanisms. Instead of using programmatic APIs to use these mechanisms, they use declarative mechanisms, such as XML metadata. Thus, application developers are required to specify the middleware-specific QoS mechanisms in XML.

2.2 DRE system case study

Determining which RTCORBA mechanisms and their configurations will provide the most effective approach to realize domain-specific QoS policies is a hard problem faced by DRE developers. We use a simple DRE use case to highlight these challenges.

The Basic Single Processor (BasicSP) shown in Figure 3 is a scenario from the Boeing Bold Stroke component avionics computing product line [25]. BasicSP uses a publish/-subscribe service for event-based communication among its components, and has been developed using a QoS-enabled component middleware platform. The application is deployed using a single deployment plan on two physical nodes.

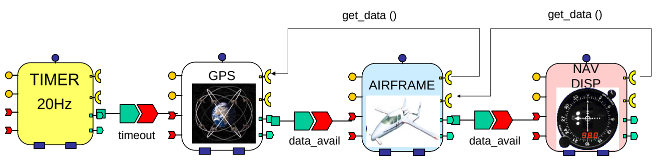

Fig. 3: Basic single processor

As shown in Figure 3, a GPS device sends out periodic position updates to a GUI display that presents these updates to a pilot. The desired data request and the display frequencies are at 20 Hz. The scenario begins with the *GPS* component being invoked by the *Timer* component. On receiving a pulse event from the *Timer*, the *GPS* component generates its data and issues a data available event.

The *Airframe* component retrieves the data from the *GPS* component, updates its state, and issues a data available event. Finally, the *NavDisplay* component retrieves the data from the *Airframe* and updates its state and displays it to the pilot. In its normal mode of operation, the *Timer* component generates pulse events at a fixed priority level, although its real-time configuration can be easily changed such that it can potentially support multiple priority levels.

It is necessary to carefully examine the end-to-end application critical path and configure the system components correctly such that the display refresh rate of 20 Hz may be satisfied. In particular, the latency between *Airframe* and *NavDisplay* components needs to be minimized to achieve the desired end goal. To this end, several characteristics of the BasicSP components are important and must be taken into account in determining the most appropriate QoS configuration space.

For example, the *NavDisplay* component receives update events only from the *Airframe* component and does not send messages back to the sender, *i.e.*, it just plays the role of a client. The *Airframe* component on the other hand communicates with both the *GPS* and *NavDisplay* components thereby playing the role of a client as well as a server. Various QoS options provided by the target middleware platform (in case of BasicSP, it is RTCORBA-based) ensure that these application level QoS requirements are satisfied.

2.3 Middleware Configuration Challenges

We now discuss why mapping the domain-specific QoS policies into middleware QoS configurations is challenging for DRE system developers.

Challenge 1. Inherent complexity in translating QoS policies to QoS configuration options. Translating QoS policies into QoS configuration options is hard because it must transform semantics from the application domain to the semantics of the underlying component middleware. QoS-enabled component middleware like CIAO leverage RTCORBA mechanisms to configure (1) *processor resources*, such as portable priorities, end-to-end priority propagation, thread pools, distributable threads and schedulers, (2) *communication resources*, such as protocol properties and explicit binding of connections, and (3) *memory resources*, such as buffering of requests.

To translate the domain-specified QoS policies into middleware QoS configurations, DRE system developers need a thorough understanding of the underlying middleware platforms. For example, in the BasicSP scenario assume that the air frame component has multiple other sensors like the GPS publishing events at different priorities. Developers must be able to determine the right request handling model to handle priorities, and determine the right concurrency model including whether lanes must be used, how many of them should be created, how many should be created statically versus dynamically, among many other decisions. DRE developers are seldom experts in low level details of the middleware.

Challenge 2. Satisfying pre/post conditions and invariants in QoS configuration. Even if a DRE developer were to translate the QoS policies into a subset of QoS configuration options, it is also necessary for them to understand the pre-conditions, invariants, and post-conditions of the different QoS configuration options since these conditions affect middleware behavior. For example, to create a thread pool with lanes in a server component, the following (non-exhaustive) pre-/post-conditions and invariants must be satisfied:

- *Pre-conditions.* A real-time POA created within a real-time ORB must be available, and the range of priorities (for the different lanes) and the type of priority mapping scheme chosen must be compatible, *i.e.*, within the limits.
- *Post-conditions.* A thread pool-with-lanes corresponding to the different priorities, along with the requested number of static (pre-defined) threads must be made available for use.
- *Invariants.* The real-time ORB should be able to match incoming request priorities to the corresponding lanes, and must always handle incoming requests for higher priority lanes before incoming requests for lower priority lanes.

This problem is exacerbated by the plethora of options and choices of valid values for each option, as well as by the fact that choosing one value for a particular option may have side effects on other options. These side effects are sometimes manifested as overt failures, such as failure to perform a mapping of CORBA priority to the underlying OS priority because of insufficient priorities in the OS to support the choice of priority mapping scheme, *e.g.*, *direct* mapping. They may also be manifested, however, as hard-to-reproduce and/or hard-to-debug runtime failures that only emerge during field testing, or after deployment, which are much harder to detect and fix.

Challenge 3. Resolving dependencies between QoS configuration options. Even with a thorough understanding of middleware QoS configuration options, manually mapping policies to configurations of the middleware cannot scale as the number of entities to configure increases. This problem stems from dependencies between the different QoS configuration options of each component, such as the dependency between the CORBA priority of a component, the chosen priority mapping scheme (to map CORBA priority to native OS priority), and the priority-banded connections policy (which selects the appropriate connection to route requests based on the request invocation priority).

If the components are connected, the side effect of the connection between components may also induce an inter-component option dependency. Since these dependencies can grow quadratically, it is infeasible for developers to manage these dependencies manually. In DRE systems with many components, the effects of changing a QoS configuration option on a component may affect many other directly connected components, their connected neighbors and so on. These dependencies can rapidly degenerate into a very large number of QoS configurations. Keeping track of dependencies between options and propagating the changes in one option to all options affected by that change is critical during the QoS configuration phase.

For example, the BasicSP scenario is a reusable component assembly that can be part of a larger workflow of components. Moreover, although the BasicSP workflow is reusable, it is conceivable that in different use cases, the components such as the timer may operate with different frequencies leading to different middleware configurations. As the number of components increases, the number of inter-component dependencies increases proportionally.

Verifying the correctness of the values of the different QoS configuration options in isolation and together with connected components is critical to the successful deployment and ultimately the operation of DRE systems. Once again, it is hard to verify the correctness of these values without automated tool support. Depending on the frequency of changes made to the domain-specified QoS policies during the development process, relying simply on empirically validating a change in QoS configuration options becomes time consuming at this scale, which slows down the design process considerably and permits subtle and pernicious errors to occur. What is needed is an automated tool support that can assure an application's evolution throughout its entire lifecycle.

The remainder of this paper shows how QUICKER helps to address these challenges.

3 The QUICKER QoS Configuration Process

Figure 4 shows the automated middleware QoS configuration approach adopted by QUICKER to address the challenges outlined in Section 2.3. QUICKER uses model-to-model transformation techniques [4] to translate the platform-independent specifications of domain-specific QoS policies modeled in one DSML into middleware-specific QoS configurations modeled in another DSML.

The source models of domain-specific QoS policies are modeled using a DSML is called *Platform-Independent Component Modeling Language* (PICML) [2]. PICML provides a construct called *Policies* that enables developers of component-based DRE systems to annotate component-based application models developed in PICML with QoS policies. These policies are specified at a higher-level of abstraction using *platform-independent* artifacts rather than using low-level, platform-specific configuration options typically found in middleware configuration files. QUICKER thus allows flexibility in mapping the same QoS policy to other middleware technologies.

QUICKER's automated mapping process transforms the PICML QoS policy models into models of middleware-specific QoS configuration options that belong to a DSML called the *Component QoS Modeling Language* (CQML). In this paper, CQML enables the modeling of platform-specific QoS

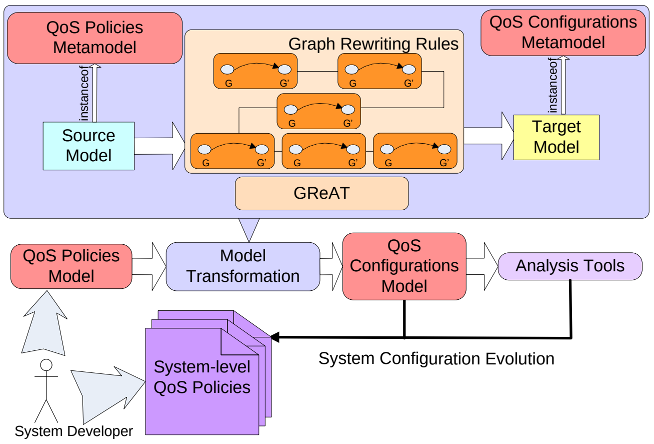

Fig. 4: **Model-driven QoS configuration process**

configurations provided by the CIAO [5] real-time CORBA Component Model middleware.

To demonstrate the viability of the QUICKER approach, we have used the Generic Modeling Environment (GME) [15] framework for developing the source and target DSMLs. GME is a metaprogrammable modeling environment with a general-purpose editing engine, separate view-controller GUI, and a configurable persistence engine. Since GME is metaprogrammable, the same environment used to define a DSML is also used to build models, which are instances of the DSML metamodel. GME provides the ability to use OCL to assign semantics to the concrete syntax of a DSML. GME also enables associating generators with individual DSMLs through a plugin mechanism. These generators are typically model interpreters that can traverse the entities of the DSML and can be implemented to synthesize different artifacts, such as XML descriptors, source code, or input to analysis tools.

The model-to-model transformations have been developed [10] using the Graph Rewriting And Transformation (GReAT) [8] framework. GReAT, which is implemented within the framework of GME, can be used to define transformation rules using its visual language, and executing these transformation rules for generating target models using the GReAT execution engine (GR-Engine).

3.1 Modeling QoS Policies: The Source DSML

QUICKER defines a `Policy` modeling construct as a generalization of QoS policies. As shown in Figure 5, source elements `Component`, `ComponentAssembly` or `Port` connections can be associated with a `Policy` element. The modeling abstractions in QUICKER allow association of multiple source elements with the same `Policy` as long as those source elements are of the same *type*. Moreover a `ComponentAssembly`'s `Policy` is also associated with all the components contained in that `ComponentAssembly`. Such associations provide significant benefits in terms of flexibility in the creation of QoS policies models and scalability of the models. The metamodels we describe below have been integrated with PICML using these associations; thus a single model of DRE system captures its entire QoS policies specification.

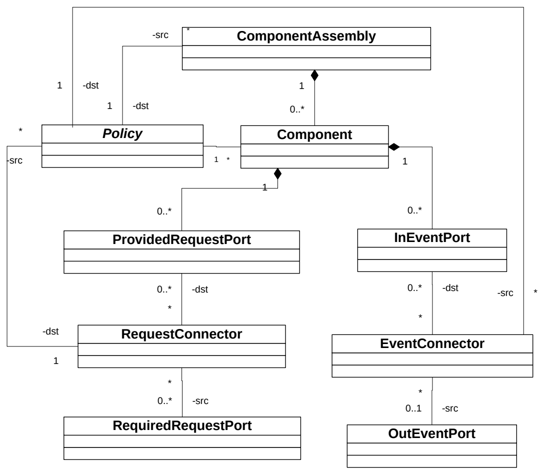

Fig. 5: **Simplified UML notation of QoS Policies associations in QUICKER**

Next we discuss the policies specification across the following two real-time QoS dimensions: (1) RT request/response that is used to specify policies for components and the synchronous connections between components, and (2) RT publish/subscribe service that is used to specify policies for asynchronous event-based connections between components.

Request/Response QoS policies. The request/response policies have component-level granularity. A `RTPolicy` element, which is derived from `Policy`, captures real-time policies of a component and may have the following two attributes: (1) `fixed_priority_service_execution`, a server component Boolean property for specifying whether or not it modifies client service invocation priorities, and (2) `bursty_client_requests`, a server component Boolean property for specifying the profile of service invocations made by its client components. Figure 6 illustrates the relevant concrete syntax for modeling the policies for the request/response real-time communication.

Publish/subscribe QoS policies. We have modeled the policies for real-time publish/subscribe event service to enable specification of QoS across asynchronous and anonymous interactions in component-based DRE systems. In the context of a publish/subscribe service, a **Subscriber** component subscribes to receive events from a **Publisher** component that generates events. Publisher (subscriber) component connects to a mediator entity, an **Event Channel**, to publish (subscribe to) events. Figure 7 illustrates the relevant concrete syntax for modeling the policies of the event channel.

A `ECPolicy` element is derived from `Policy`. It models the properties of the event channel and can be used to specify the following QoS policies: (1) `network_quality`,

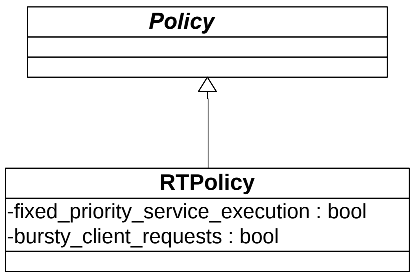
Policy
RTPolicy
-fixed_priority_service_execution : bool
-bursty_client_requests : bool

Fig. 6: **Simplified UML notation for Real-time Request/Response Policies**

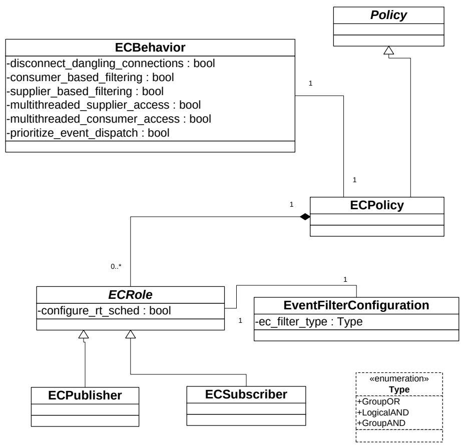
Policy
ECBehavior
-disconnect_dangling_connections : bool
-consumer_based_filtering : bool
-supplier_based_filtering : bool
-multithreaded_supplier_access : bool
-multithreaded_consumer_access : bool
-prioritize_event_dispatch : bool
ECPolicy
ECRole
-configure_rt_sched : bool
EventFilterConfiguration
-ec_filter_type : Type
ECPublisher
ECSubscriber
«enumeration»
Type
+GroupOR
+LogicalAND
+GroupAND
1
0..*

Fig. 7: **Simplified UML notation for Policies of Event Channels**

a connection-level property that captures the quality value of network used for running the application; (2) `connection_frequency`, a component-level property specifying the frequency at which the component (dis)connects with the publish/subscribe connection; (3) `event_distribution_ratio`, a connection-level property that specifies the ratio: $\frac{E^a{}_c}{E^s{}_c}$, where $E^a{}_c$ denotes number of events available for subscription at connection c and $E^s{}_c$ denotes average number of events subscribed to at connection c by each subscriber component.

These modeling capabilities are at a sufficiently high level of abstraction and are intuitive to be applied to a variety of publish/subscribe mechanisms. All the policies have an enumerated data type with values `LO` and `HI`.

3.2 Modeling Middleware QoS Options: The Target DSML

Rather than directly transforming source models of DRE system QoS policies into configuration descriptors in XML required for deploying it on the middleware, we chose to generate models of middleware-specific QoS options from these source models such that they can be used for further analysis such as model-checking QoS properties of the DRE system.

This represents the Component QoS Modeling Language (CQML). We have developed interpreters for parsing CQML system models and generating XML deployment descriptors in preparation for deploying the DRE system on the target environment.

Request/Response QoS options. A simplified UML notation of CQML modeling elements for Request/Response QoS options is shown in Figure 8.

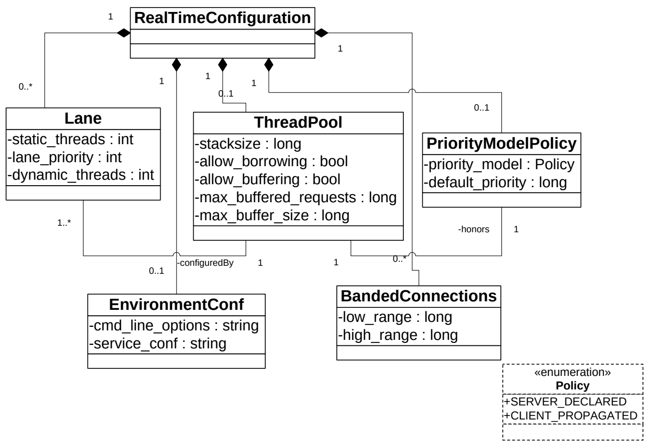

Fig. 8: **Simplified UML notation for RT-CCM Request/Response Configuration Options**

As shown, CQML defines the following elements corresponding to several RT-CCM configuration mechanisms: (1) `Lane`, which is a logical set of threads each one of which runs at `lane_priority` priority level. It is possible to configure *static* thread (*i.e.*, those that remain active till the system is running and *dynamic* thread (*i.e.*, those threads that are created and destroyed as required) numbers using `Lane` element; (2) `ThreadPool`, which controls various settings of `Lane` elements, or a group thereof. These settings include `stacksize` of threads, whether borrowing of threads across two `Lane` elements is allowed, and maximum resources assigned to buffer requests that cannot be immediately serviced; (3) `PriorityModelPolicy`, which controls the policy model that a particular `ThreadPool` follows. It can be set to either CLIENT_PROPAGATED if the invocation priority is preserved, or SERVER_DECLARED if the server component changes the priority of invocation; and (4) `BandedConnections`, which defines separate connections for individual (client) service invocations.

An example snippet of XML descriptors that are generated out of CQML models for request/response policies is shown in Listing 1. As shown in lines 3-19, `ThreadPoolWithLanes` consists of the following two options: (a) `Lane`, specifying the number of thread resources and their type, and (b) `ThreadPool`, governing various characteristics of a pool of `Lanes`. For each option in this listing, the value of that option must be enclosed in appropriate XML tags such that the QoS specification is valid and complete.

```
1   <orbConfigs>
2   <resources>
3     <threadpoolWithLanes id="threadpool-2">
4       <threadpoolLane>
5         <static_threads>5</static_threads>
6         <dynamic_threads>0</dynamic_threads>
7         <priority>2</priority>
8       </threadpoolLane>
9       <threadpoolLane>
10        <static_threads>5</static_threads>
11        <dynamic_threads>0</dynamic_threads>
12        <priority>1</priority>
13      </threadpoolLane>
14      <stacksize>0</stacksize>
15      <allow_borrowing>false</allow_borrowing>
16      <allow_request_buffering>false</allow_request_buffering>
17      <max_buffered_requests>0</max_buffered_requests>
18      <max_request_buffered_size>0</max_request_buffered_size>
19    </threadpoolWithLanes>
```

Listing 1: XML Descriptor Snippet for RT request/response Configuration in LwCCM

Publish/subscribe QoS options. For QoS configuration of asynchronous event communications, CQML defines the following elements: (1) `Publisher` and `Subscriber` modeling elements containing all the event source and sink settings, respectively. These include, for example, thread locks management mechanisms for publishers (subscribers) that are accessed by multi-threaded systems, and types of event filtering used, (2) `RTECFactory` element contains configurations specific to the event channel itself. These include, for example, event dispatching method that controls how events from publishers are forwarded to the respective subscribers, scheduling of events for delivery and other scheduler-related coordination, and handling of timeout events in order to forward them to respective subscribers, and (3) `FilterGroup` element that specifies strategies to group more than one filters together for publishers (subscribers).

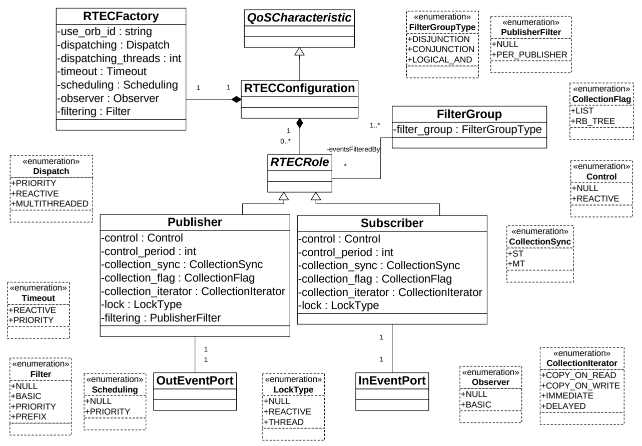

Fig. 9: **Simplified UML notation for RT-CCM Publish/-Subscribe Configuration Options**

Having a DSML such as CQML has the following advantages:

- The (generated) CQML models can be used for further analysis such as, for example, model-checking QoS properties of the DRE system.
- Using models of application at each step of software development increases traceability as QoS policies are translated to low-level QoS configurations *i.e.*, to middleware descriptors required for deploying the DRE system.
- The CQML models are *closed* in terms of modification by system developers, and can only be (re-)generated by applying QUICKER model transformations on system QoS policy models. Such a design choice simplifies software development and is a crucial step towards addressing productivity problem [11] at the middleware level.

3.3 Model-to-model Transformation Algorithms

The QUICKER model transformation rules have been defined in GReAT and are based on our past experiences in configuring QoS for component-based DRE systems. Transformation rules are defined using the GReAT visual language. They are applicable to any system model that conforms to the Policies DSML, and thus can be used by the system developers repetitively during the development and/or maintenance phase(s) of the DRE system. QUICKER model transformations preserve the granularity specified in the source models.

Figure 10 shows the high-level steps involved in developing transformation algorithms using the GReAT tool chain. In Step 1, the source and target domain-specific modeling languages (DSMLs) for the transformation tool chain are defined. In Step 2, developers use the GReAT transformation language to define various translation rules in terms of patterns[1] of source and target modeling objects. Rules that cannot be captured visually can be encoded as C++ code snippets associated with the mapping blocks. In Step 3, a source model instance is provided to the GReAT framework. Finally, in Step 4, developers execute the GReAT engine (called the GR-engine) that translates the source model using rules specified in Step 2 into the target model.

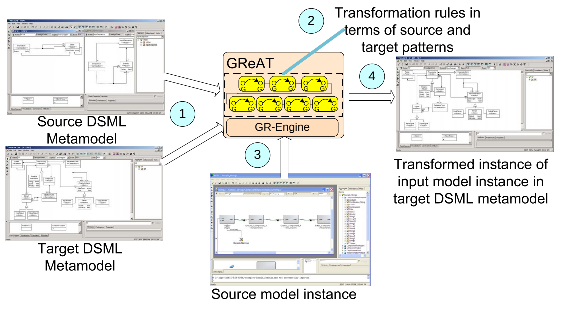
Transformation rules in terms of source and target patterns
2
GReAT
GR-Engine
1
Source DSML Metamodel
Target DSML Metamodel
3
4
Source model instance
Transformed instance of input model instance in target DSML metamodel

Fig. 10: **Model Transformation Process in GReAT (details of the models are not important for this figure).**

More specifically, the GR-engine execution involves the following steps: (1) executing the *master interpreter* that generates the necessary intermediate files containing all the rules in the current transformation, (2) compile these intermediate files, if not done already, and (3) run the generated executable.

We now describe the mapping rules we have encoded using the GReAT transformation language for mapping QoS policies into middleware QoS configurations for situations that use request/response and publish/subscribe semantics.

Mapping real-time QoS Policies. Let ${R_p}^o$ and ${R_p}^i$ denote, respectively, the set of outgoing (required/event source) and incoming (provided/event sink) ports of component $p \in P$. Let S and C be the sets of server and client components respectively and are given by:

$p \in S \; if \; {R_p}^i \neq \emptyset$ and $p \in C \; if \; {R_p}^o \neq \emptyset$

Algorithm 1 describes (non-exhaustive) RT-CCM QoS mappings in QUICKER. Lines 5-13 show the thread resource allocation scheme for server components. For every incoming port of a server component, the number of interface operations and client components are counted (lines 9 and 10).

[1] Here, pattern refers to a valid structural composition using model objects in the source (target) DSML.

These counts are used by the auxillary function *ThreadResources* to calculate the total threads required for handling all client service invocations at that server.

Algorithm 1: **Real-time QoS policies mapping**

```
   Input: set of client components $C$, set of server components $S$,
          set of bursty client components $B$, set of threadPool
          lanes $TPLanes$
1  begin
2      InterfaceOperationsCount $ioc$; ClientsCount $cc$;
3      IncomingPort $ip$; OutgoingPort $op$;ThreadCount $tc$;
4      Component $c$;set of Components $CPS$; Buffering $bf$;
5      foreach $p \in S$ do
6          $ioc \leftarrow 0$; $cc \leftarrow 0$; $tc \leftarrow 0$; $bf \leftarrow false$;
7          $CPS \leftarrow ClientComponents(p)$;
8          foreach $ip \in R_p{}^i$ do
9              $ioc \leftarrow ioc + countOperations(p, ip)$;
10             $cc \leftarrow cc + countClientComponents(p, ip)$;
11         end
12         $tc \leftarrow ThreadResources(ioc, cc)$;
13         $createTPLanes(p, tc)$;
14         foreach $c \in CPS$ do
15             if $c \in B$ then
16                 $bf \leftarrow true$;
17                 $assignThreadResources($
18                 $TPLanes_p, c, tc)$;
19             $assignTPoolAttributes(TPLanes_p, bf)$;
20             $ioc \leftarrow 0$;
21             foreach $op \in R_c{}^o$ do
22                 $ioc \leftarrow ioc + countOperations(c, op)$;
23             end
24             $createBands(c, ioc)$; $matchPriorities(p, c)$;
25         end
26     end
27 end
```

Algorithm 2: **Publish/Subscribe service QoS policies mapping**

```
   Input: set of components $CPS$
1  begin
2      Component $c$; ThreadPoolLaneCount $lc$;
3      NetworkQuality $nq$;
4      foreach $c \in CPS$ do
5          $lc = countThreadResources(c)$;
           $cf = connectionFrequency(c)$;
           $nq = networkQuality(c)$;
           $dr = eventDistributionRatio(c)$;
6          if $lc \neq 1$ then
7              $PC_c{}^s = MT$; $L_c = THREAD$;
8          else
9              $PC_c{}^s = ST$; $L_c = NULL$;
10         end
11         if $cf \neq LO$ then
12             $PC_c{}^t = LIST$; $PC_c{}^i = COPY\_ON\_READ$;
13         else
14             $PC_c{}^t = RB\_TREE$; $PC_c{}^s = COPY\_ON\_WRITE$;
15         end
16         if $nq \neq LO$ then
17             $CP_c = NULL$;
18         else
19             $CP_c = REACTIVE$;
20         end
21         if $c \in S$ then
22             if $dr \neq LO$ then
23                 $SF_c = PER\_SUPPLIER$;
24             else
25                 $SF_c = NULL$;
26             end
27     end
28 end
```

For handling bursts of client requests, server components should configure their thread pool to grow dynamically such that threads are created only when required.*assignThreadResources* function is used to adjust the ratio of static and dynamic threads for a server, depending on whether its `bursty_client_requests` property is set to TRUE. In addition, lane borrowing feature at the server is set to TRUE such that the thread pool lanes across various priority levels can be borrowed. Finally, *PriorityBands* are configured and the their priority values are matched with server-side lane values in line 24.

Mapping publish/subscribe QoS policies. Let $PC_c{}^s$ denote the synchronization mechanism, $PC_c{}^t$ denote the type, $PC_c{}^i$ denote the iterator in proxy collection PC for component c, respectively. Let L_c denote the locking policy, CP_c denote control policy, SF_c denote supplier-based filtering at component c, respectively. Algorithm 2 gives the (non-exhaustive) publish/subscribe QoS mappings.

A publish/subscribe service has several settings for configuring the way collections of publisher and subscriber object references are created and accessed, which must be chosen appropriately for individual applications. Lines 6-9 in Algorithm 2 show how the choice of serialization mechanism is affected by the number of thread resources configured at component c.

The choice of the *type* of collection is based on the following: (1) RB_TREE data structure exhibits faster ($O(log(n))$) insertion and removal operations. Therefore, it is more suited for connections whose components have a high (dis)connection rate; (2) LIST data structure on the other hand, should be chosen in cases where iteration is frequent (and therefore, more crucial for efficient application execution) than modifications to it.

Lines 11-14 give the steps in algorithm that configure the collection type. Finally, REACTIVE policy is chosen for applications that use low-quality value network on Lines 16-19, which ensures that (publisher/subscriber) components are periodically polled for determining their states (*i.e.*, whether or not they are connected to the event channel).

3.4 Resolution of Challenges 1 & 3

Target typed graph elements (*i.e.*, QoS options), are well-understood by middleware implementation experts. We expect that the QUICKER transformation algorithms 1 and 2 will be designed in terms of source and target typed graphs by these experts. DRE system developers will only need to think of their policies at levels that are intuitive. Hence, they can describe their system QoS policies using the modeling capabilities discussed in Section 3.1.

By providing platform-independent modeling elements in QUICKER and defining representational semantics that closely follow those of the system policies, QUICKER allows system developers to describe system QoS using simple, intuitive notations. Further, model transformations defined in QUICKER automatically identify and deduce QoS configurations that are best suited to achieve the desired QoS for DRE systems being configured. This resolves Challenge 1 described in Section 2.3.

Challenge 3 described in Section 2.3 is partly resolved as follows. QUICKER transformation rules contain information about the semantics of the QoS options, their inter-dependencies, and how they affect the high-level QoS policies of a DRE system and therefore are used to assign values to the subset of options chosen earlier. Further, QoS options semantics are known precisely during transformations, and thus QUICKER ensures preservation of the target typed graph semantics. Component interactions defined in input typed graph instance (*i.e.*, source model), along with the user-specified QoS policies captured in that instance are used to completely generate an instance of the output graph.

4 Correctness of the Model Transformation Process and Generated Configurations

In any approach that provides automation, two key assurances must be provided: (1) that the automation process itself is correct, and (2) that the artifacts produced as a result of the automation are correct. In our approach, it is therefore necessary to provide an assurance that the process of mapping the QoS policy specifications into middleware-specific QoS configurations is correct, and that the generated mappings themselves are correct including all the dependencies between the mappings.

This task entails ensuring the correctness of the QoS mapping process, *i.e.*, providing assurance that the QoS configurations generated are equivalent to the QoS policies from which these options are mapped. In our case, this translates to verification of the QoS transformations used, and (2) correctness of the generated QoS options themselves, *i.e.*, whether the configuration parameters themselves are correct and whether the individual values of these options are appropriate locally (*e.g.*, for a component) as well as globally (*e.g.*, for all dependent components).

4.1 Verifying the Correctness of the Transformations and Generated Artifacts

The difficulty in verifying the correctness of model transformations is akin to the difficulty in verifying compilers for high-level languages. This difficulty can be alleviated to some extent if we restrict the verification of the transformation process to the verification of instances of the transformations. In other workds, a transformation may be said to have executed correctly if a certain instance of its execution produces an output model that preserves certain properties of interest. We designate such an instance as certified correct.

In our approach, we have used a transformation verification technique based on structural correspondence [17, 18]. The intuition behind this approach is as follows: Specific structural configurations in the source model (such as the QoS policies in PICML) produce specific structural configurations in the target model (such as CIAO/RTCORBA-specific QoS configurations in CQML). The rules to accomplish the structural transformations may be simple or complicated. However, it is fairly straightforward to compare and verify that the correct structural transformation was made if we already know which parts of the source structure map to which parts of the target structure.

In our technique to verify a transformation by structural correspondence, we first define a set of structural correspondence rules specific to a certain transformation, which are effectively a relation between elements of the source model and that of the target model. We then use cross-links to trace source elements with the corresponding target elements, and finally use these cross-links to check whether the structural correspondence rules hold. In essence, we expect that the correspondence conditions are independently specified for a model transformation, and an independent tool checks if these conditions are satisfied by the instance models, after the model transformation has been executed.

Figure 11 illustrates the verification technique. The source and target portions of the transformation are treated as typed, attributed graphs, called the source and target metamodels. The correctness of the transformation is specified as a relation between these graphs. Such a relation, called a structural correspondence, is specified by identifying pivot nodes in the metamodels and specifying what constitutes a correct transformation for these nodes. The existing transformation rules written in the GReAT visual language are extended to generate annotations along with the instance models. Once this setup has been completed, every execution of the transformation is followed by an automatic correctness checker, which uses the generated annotations to check if the correctness conditions were satisfied for the generated output model. Recall from Section 3.3 that the GR-Engine in GReAT is responsible for executing the transformations and hence also the correctness checking. If this check is satisfied, we can say that the output model instance is ‘certified’ correct.

Using structural correspondence, the verification consists of two phases: the specification of the correctness conditions, and the evaluation of the correctness. In the first phase, we identify important points in the transformation, and specify the structural correspondence rules for these points. This is the responsibility of the transformation developer to specify these rules. From these rules, a model traverser is automatically generated, which will traverse and evaluate the corre-

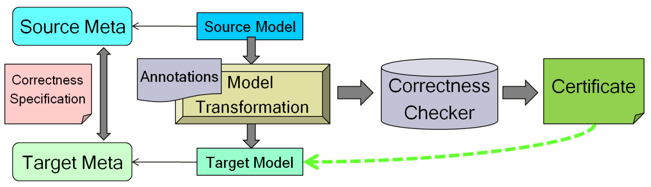

Fig. 11: **Verifying model transformations**

spondence rules on the instance models. This step needs to be performed only once.

The second phase involves invoking the model traverser after each execution of the model transformation. In this phase, the model instance being transformed is traversed, and the structural correspondence rules are evaluated at each relevant node. If any of the rules are not satisfied, it indicates that the model has not been transformed satisfactorily.

Structural correspondence rules are described using (1) specification of the correspondence condition itself, and (2) the rule path expressions, which are similar to XPath queries. Figure 12 shows how we have used cross-links in GReAT as means of specifying the correspondence condition between input and output language objects such that their equivalence can later be established. In the figure we show the composite metamodel that relates an element from PICML (in this case the QoS policies for a request/response scenario) with an element from CQML (in this case the request handling priority model supported in RTCORBA).

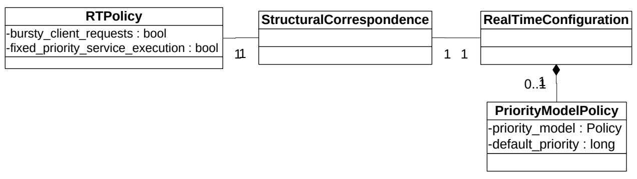

Fig. 12: **Structural correspondence using cross-links**

`RTPolicy` is an input language object that denotes real-time policy specification for a component. It has a correspondence relation with `RealTimeConfiguration` output language object indicated by the presence of a cross-link between them in Figure 12. Additionally, one of the transformation rules in our QoS mapping algorithms states that if the Boolean attribute `fixed_priority_service_execution` of `RTPolicy` is set to TRUE in the input model, then `priority_model` attribute of `PriorityModelPolicy` object be set to SERVER_DECLARED in the output model. Otherwise `priority_model` should be set to CLIENT_PROPAGATED.

Moreover, if `priority_model` is set to SERVER_DECLARED for a component, `Lane` values at that component and `BandedConnection` values at its clients must match. In order to complete the correspondence rule specification, the above is encoded as a rule path expression as follows:

```
(RTRequirement.
fixed_priority_service_execution = true ∧
( ∀ b ∈ RTConfiguration. BandedConnection
∃ l ∈ RTConfiguration. Lanes :
    (b.low_range ≤ l.priority ≤ b.high_range)) ∧
RealTimeConfiguration.PriorityModelPolicy.
priority_model = "SERVER_DECLARED") ∨
(RTRequirement.
fixed_priority_service_execution = false ∧
RealTimeConfiguration.PriorityModelPolicy.
priority_model = "CLIENT_PROPOGATED")
```

If this expression evaluates to TRUE on an instance model, then it implies that the QoS configuration for this particular property has been mapped correctly. This applies to the `RTRequirements` and `RealTimeConfiguration` classes, and correspondence condition is added as a link between these classes in the metamodel. Similar to correspondence condition between `RTRequirements` and `RealTimeConfiguration` we described, other conditions for each of the QoS mapping rules have been specified ensuring that the transformation is verified correct if all these conditions are satisfied.

Note that the transformation verification does not imply the correctness of the QoS mapping rules themselves – rather it provides an assurance that the model transformation specification and implementation correctly mapped the QoS specifications that were formulated before.

4.2 Verifying the Generated QoS Configurations

QUICKER uses a novel approach to check the correctness of the generated configurations. QUICKER achieves this capability by reducing the problem to a model checking problem. This section illustrates how the correctness of QoS configuration mappings is verified using the Bogor model-checking framework, which is a customizable explicit-state model checker implemented as an Eclipse plugin.

Verifying a system using Bogor involves defining (1) a model of the system using the *Bogor Input Representation* (BIR) language and (2) the *property* (*i.e.*, specification) that the model is expected to satisfy. Bogor then traverses the system model and checks whether or not the property holds. To validate QoS configuration options of an application using Bogor, we need to specify the application model and its QoS configurations. We use Bogor's extension features to customize the model-checker for resolving the QoS configuration challenges for component-based applications.

It is cumbersome to describe middleware QoS configuration options using the default input specification capabilities of BIR. This is because such a representation is at a much lower level of abstraction compared to domain-level concepts, such as components and QoS options, which we want to model-check. Additionally, specifying middleware QoS configuration options using BIR's low-level constructs can yield an unmanageably large state space since representing domain-level concepts with a low-level BIR specification requires additional auxiliary states that may be irrelevant to the properties being model-checked [23]. Therefore we have defined composite language constructs that represent functional sub-systems (such as components) and QoS options (such as thread pools) as though they were native BIR constructs.

Listing 1 shows an example of our QoS extensions in Bogor to represent QoS configuration options in middleware, which define two new data types: `Component`, which corresponds to a middleware component such as in our CIAO middleware, and `QoSOptions`, which captures QoS configuration options, such as `lane`, `band`, and `threadpool`.

```
extension QoSOptions for
edu.ksu.cis.bogor.module.QoSOptions.QoSOptionsModule
{
  // Defines the new type to be used for
  typedef lane;
  typedef band;
  typedef threadpool;
  typedef prioritymodel;
  typedef policy;
  // Lane constructor.
  expdef QoSOptions.lane createLane (
   int static, int priority, int dynamic);
  // ThreadPool constructor.
  expdef QoSOptions.threadpool
  createThreadPool (boolean allowreqbuffering,
   int maxbufferedrequests, int stacksize, int
   maxbuffersize, boolean allowborrowing);
  // Set the band(s) for QoS policy.
  actiondef registerBands (QoSOptions.policy
   policy, QoSOptions.band ...);
  // Set the lane(s) for QoS policy.
  actiondef registerLanes (QoSOptions.policy
   policy, QoSOptions.lane ...);
  ...
}
extension Quicker for
edu.ksu.cis.bogor.module.Quicker
{
  // Defines the new type.
  typedef Component;
  // Component Constructor.
  expdef Quicker.Component
  createComponent (string component);
  // Set the QoS policy for the component.
  actiondef registerQoSOptions (Quicker.Component
    component,QoSOptions.policy policy);
  // Make connections between components.
  actiondef connectComponents (Quicker.Component
    server,Quicker.Component client);
  ...
}
```

Listing 1: **QUICKER BIR extension**

In addition to defining constructs that represent domain concepts, such as components and QoS options, we also need to specify the *property* that the application should satisfy. In our case, a property simply means whether or not the QoS configurations are verified correct. Thus, since we need to verify the values of various QoS options as a means to check whether the application property is satisfied, we define *rules* that capture values of these QoS options. BIR primitives are used to express these rules in the input specification of DRE system. Primitives are also used to capture component interconnections in BIR format which are required for populating the dependency structure for the specified input application. They are also used later during verification of options for connected components. We demonstrate this capability in the context of a representative system in Section 4.3.

Applications that need to be model-checked by Bogor must be represented in BIR format. Writing and maintaining BIR manually can be tedious and error-prone for domain experts (*e.g.*, avionics engineers) since configuring application QoS policies is typically done iteratively. Depending on the number of components and available configuration options, manual processes do not scale well.

To automate the process of creating BIR specification of applications, we therefore used the generative capabilities in GME to automatically generate BIR specification of an application from its QoS configurations model. This generative process is done in GME using a model interpreter that traverses the QoS configurations model and generates a BIR file that captures the application structure and its QoS properties. Our toolchain therefore automates the entire process of mapping application QoS policies to middleware QoS options, as well as converting these QoS options into BIR. A second model interpreter is used to generate the Real-time CCM-specific descriptors required to configure functional and QoS properties of an application and deploy it in its target environment. In the next section we empirically validate these generated QoS configurations.

4.3 Validating BasicSP QoS Configuration Dependencies via Model-checking

The QoS extensions to the BIR format described in Section 4.2 are used in maintaining and resolving dependencies between application components. For example, consider a real-time configuration of BasicSP scenario in which each of the *GPS*, *AirFrame*, and *NavDisplay* components are configured to have `priority bands` for separate service invocation priorities and the *Timer* component is configured to support multiple priority levels during generation of pulse events. Given such a configuration, we have that `priority band` values at *GPS* (client) component must match `ThreadPoolLanes` at *Timer* (server) component *i.e.*, a direct configuration dependency exists between these two components.

Further, since the pulse events are subsequently reported to *AirFrame* and *NavDisplay* components there is a similar indirect dependency between `band` values at these components and `lanes` at *Timer* component. The dependency structure of BasicSP scenario is maintained in QoS extensions to track such dependencies between QoS options.

Figure 13 represents the dependency structure generated using QoS extensions with the given configurations for our BasicSP scenario. Occurrences of change in configurations of either of the dependent components are followed by detection of potential mismatches such that all dependencies are exposed and resolved during application QoS design iterations.

A similar approach can be carried out if the DRE system has multiple different modes of operation, which in turn

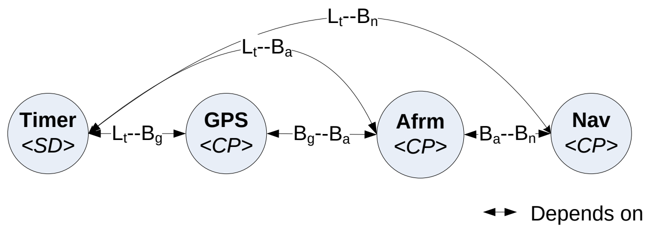

Fig. 13: **Dependency structure of BasicSP. L_c denotes threadpool lane and B_c denotes priority bands at component *c*.** SD **and** CP **indicate the** SERVER_DECLARED **and** CLIENT_PROPAGATED **priority models, respectively.**

define multiple different regions of operation. Each such operating region may then have a set of configuration options. The Bogor-based model checking approach can now verify the dependencies among options for connected components for multiple modes of operation. The verified sets of configurations can then be used by the middleware to seamlessly adapt from one set to another depending on the mode.

4.4 Resolution of Challenges 2 and 3

Challenge 2 from Section 2.3 is resolved by the assurance of correctness provided by structural correspondence as follows. Pre- and post-conditions can be satisfied by assuring that the generated configurations are indeed the expected artifacts. Since the transformations are automated, developers need not worry about the plethora of configuration options and their interactions. Not all invariants may be satisfied by the transformation process alone. For example, whether the ORB can eliminate or minimize priority inversions depends solely on the quality of the middleware implementation and the OS platform on which it runs. To address this shortcoming, a potential solution may require that the transformation algorithms be enhanced to incorporate features of the available platform-specific support, such as the real-time features provided by the OS. We do not provide this capability in this paper.

Finally, the model-checking extensions ensure that the system QoS configuration is valid at the local- and global-level including all the dependencies. The entire process is automated in QUICKER and thus, can be repeated when changes in QoS policies of an application occur. For example, QUICKER helps DRE systems to evolve by automatically (re-)calculating the dependencies between options, and can thus be used repeatedly by DRE developers during the entire lifecycle thereby addressing Challenge 3.

Note that QoS configuration challenges also arise during runtime. We do not expect model checking to be applied at runtime. Rather, in the context of DRE systems, system developers will formally define the regions of operation of the system and the desird QoS policies as system conditions vary. Accordingly QUICKER can synthesize and verify the configurations for all the operating regions. The middleware is then subsequently responsible for adapting to the new configurations as the operating region of the system changes. QUICKER guarantees *a priori* availability of QoS configurations that are tailored to each operating region.

5 Related Work

We present related work comparing and contrasting them with QUICKER.

Validation and Analysis Techniques. Model-driven techniques in [13,27] rely on a visual interface to help developers select a wide array of middleware QoS options for their applications. Such information is later used for generating test suites for purposes of empirical evaluation. In contrast, our configuration process does not expose the developers to all of the configuration space of underlying middleware and relies on platform-specific heuristics for generating QoS configurations. Further, using our process, the correctness of generated configurations is established in the design time. We argue that since our transformation algorithms codify best practices and patterns in middleware QoS configuration, QoS design and evolution throughout the system lifecycle using our approach is faster.

Analysis tools such as VEST [26], Cadena [6] and AIRES [12] evaluate whether certain timing, memory, power, and cost constraints and functional dependencies of real-time and embedded applications are satisfied. Our configuration process can be used as a complementary QoS design and analysis technique to these efforts since it emphasizes on mechanisms to (1) translate design-intent into actual configuration options of underlying middleware and (2) verify that both the transformation and subsequent modifications to the configuration options remain semantically valid.

QoS Design and Specification Techniques. The Adaptive Quality Modeling Language (AQML) [19] provides QoS adaptation policy modeling artifacts. AQML generators can (1) translate the QoS adaption policies (specified in AQML) into Matlab Simulink/Stateflow models for simulations using a control-centric view of QoS adaptation and (2) generate Contract Definition Language (CDL) specifications from AQML models to be used in target middleware. Our work differs with AQML since middleware model in QUICKER precisely abstracts the actual real-time CORBA implementation and does not need a two-level declarative translation (from AQML to CDL to target middleware) to achieve QoS configuration.

Ritter *et.al.* [22] describe CCM extensions for generic QoS support and discuss a QoS metamodel that supports domain-specific multi-category QoS contracts. The work in [7] focuses on capturing QoS properties in terms of *interaction patterns* among system components that are involved in executing a particular service and supporting run-time monitoring of QoS properties by distributing them over components (which can be monitored) to realize that service. Another approach that uses an aspect-oriented specification technique for component-based distributed systems is discussed in [3]. This work deals with specification of functional behavior, non-functional behavior, QoS management policies, and requirements of the application and synthesis of QoS management components for that supporting application-level adaptation strategies.

In contrast to the projects and tools described above, our work focuses on automating the error-prone activity of mapping platform-independent QoS policies to middleware-specific QoS configuration options. Representing QoS policies as model elements allows for a unified (with functional aspects of the application) and flexible QoS specification mechanism, while automating evolution of the QoS policies with application evolution; the platform-independent QoS policies also allow configurable re-targeting of the QoS mapping to support other types of middleware technologies.

6 Concluding Remarks

With the trend towards implementing key DRE system infrastructure at the middleware level, achieving the desired QoS is increasingly becoming more of a configuration problem than a development problem. The flexibility of configuration options in QoS-enabled component middleware, however, has created a new set of challenges. Key challenges include determining the correct set of values for the configuration options, understanding the dependency relations between the different options, and evolving the QoS configurations with changes to application functionality.

In this paper, we discussed how model transformations provided by our QUICKER tool automates the mapping of DRE system QoS policies into middleware-specific QoS configuration options. We showed how structural correspondence between input and output languages in our model-driven approach can be used to establish that initial system requirements are correctly mapped to middleware QoS options. We verified the correctness of generated QoS options using a model-checker and empirically showed that they are effective in satisfying system requirements.

We demonstrated the ideas behind QUICKER concretely in the context of the CIAO Lightweight CORBA Component Model middleware. By combining model transformation and generative techniques with advanced model-checking technologies, QUICKER automates the mapping of QoS policies of applications to QoS configuration options for the CIAO middleware technology. However, QUICKER's separation of platform-independent and platform-dependent concerns enables the use of PICML models, which is our source DSML, to specify QoS policies that can be mapped to other types of middleware, such as Web Services and EJB.

As a result, developers can concentrate on inherent complexities in the application domain rather than wrestle with low-level middleware-specific configuration options. QUICKER also helps ensure the validity of the values for the QoS configuration options, both at the individual component (local) level and at the aggregate application (global) level.

The following is a summary of lessons learned from our experience in using QUICKER:

• **QoS mapping is critical to successful deployment of systems built using component middleware.** With the increase in configuration complexity, the QoS mapping capabilities provided by QUICKER are essential to managing the complexity. Configuration of middleware options to achieve the desired QoS in DRE systems can be viewed as an directed acyclic graph whose root is the high-level mission requirements, edges are the individual mappings joining the vertices in a top-down fashion, and the vertices correspond to the different options available at each intermediate layer of abstraction. QUICKER is a part of a chain of mappings starting from high-level mission requirements to the actual deployment platform, and resides between the application components and the underlying component middleware implementation.

By employing DSMLs, QUICKER not only simplifies the QoS mapping process for DRE system developers, it also preserves the rich semantics associated with the mapping between the QoS policies and QoS configuration options at this level. Using such tools also helps QUICKER integrate with mapping technologies that exist both *above* (*e.g.*, mission requirement mapping tools, functional decomposition tools, and functional analysis tools) and *below* (*e.g.*, deployment planning tools) the level at which QUICKER operates in a component-based DRE system development lifecycle.

• **Horizontal mapping of QoS is as important as vertical QoS mapping.** QUICKER currently focuses on mapping application QoS policies onto a single underlying middleware technology. Large-scale DRE systems—particularly systems requiring dynamic resource management [14]—are often composed of heterogeneous technologies. It is therefore essential for QoS mapping tools to not only support *vertical mapping* (*i.e.*, the mapping of policies and validation onto a single technology) but also *horizontal mapping* (*i.e.*, the mapping of QoS policies onto multiple heterogeneous technologies, while reconciling the differences between these technologies). Until such mapping is performed, QoS configuration and associated tools will remain as islands, which significantly complicates integration efforts for large-scale DRE systems. Care must be taken, however, to not introduce additional learning curve for developers.

In the future in order to show its scalability we plan to apply and evaluate our technique on complex and large-scale DRE systems. Our current approach is one-dimensional *i.e.*, both the QoS requirements mapping and configuration validation is done for a single dimension (such as real-time request-response or publish-subscribe communication dimensions). In the future we plan to investigate and develop configuration techniques under simultaneous requirements across distinct QoS dimensions. An effort is underway in extending our process for other component middleware platforms that exhibit the same level of configurability. As part of this effort, we are looking at development of parameterized model transformations that allow specification of templatized QoS mappings and later generation of platform-specific QoS mapping instances by specializing these templatized mappings. These parametrized models are, however, unlike the intermediate representations used in related efforts, such as AQML [19].

The QUICKER tool is available as open-source from www.dre.vanderbilt.edu/CoSMIC/.

References

1. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In K. G. Larsen and P. Niebert, editors, *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2003.
2. K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 190–199, Washington, DC, USA, 2005. IEEE Computer Society.
3. L. Blair, G. S. Blair, A. Anderson, and T. Jones. Formal Support For Dynamic QoS Management in the Development of Open Component-based Distributed Systems. *IEEE Software*, 148(3), Nov. 2001.
4. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, 2006.
5. G. Deng, C. Gill, D. C. Schmidt, and N. Wang. QoS-enabled Component Middleware for Distributed Real-Time and Embedded Systems. In I. Lee, J. Leung, and S. Son, editors, *Handbook of Real-Time and Embedded Systems*. CRC Press, 2007.
6. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of the 25th International Conference on Software Engineering*, pages 160–172, Portland, OR, May 2003.
7. Jaswinder Ahluwalia and Ingolf H. Krger and Walter Phillips and Michael Meisinger. Model-Based Run-Time Monitoring of End-to-End Deadlines. In *Proceedings of the Fifth ACM International Conference On Embedded Software*, pages 100–109, Jersey City, NJ, Sept. 2005. ACM.
8. 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. www.jucs.org/jucs_9_11/on_the_use_of.
9. A. Kavimandan, K. Balasubramanian, N. Shankaran, A. Gokhale, and D. C. Schmidt. Quicker: A model-driven qos mapping tool for qos-enabled component middleware. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 62–70, Washington, DC, USA, 2007. IEEE Computer Society.
10. A. Kavimandan and A. Gokhale. Automated Middleware QoS Configuration Techniques using Model Transformations. In *Proceedings of the* 14^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 93–102, St. Louis, MO, USA, Apr. 2008.
11. A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture*(MDA^{TM})*: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, Apr 2003.
12. S. Kodase, S. Wang, Z. Gu, and K. G. Shin. Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-time Systems using Shared Buffers. In *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS 2003)*, Washington, DC, May 2003. IEEE.
13. A. S. Krishna, E. Turkay, A. Gokhale, and D. C. Schmidt. Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems. In *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, pages 180–189, San Francisco, CA, Mar. 2005. IEEE.
14. P. Lardieri, J. Balasubramanian, D. C. Schmidt, G. Thaker, A. Gokhale, and T. Damiano. A Multi-layered Resource Management Framework for Dynamic Resource Management in Enterprise DRE Systems. *Journal of Systems and Software: Special Issue on Dynamic Resource Management in Distributed Real-time Systems*, 80(7):984–996, July 2007.
15. Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.
16. M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
17. A. Narayanan and G. Karsai. Towards verifying model transformations. *Electronic Notes on Theoretical Computer Science*, 211:191–200, 2008.
18. A. Narayanan and G. Karsai. Verifying model transformations by structural correspondence. *Electronic Communications of the EASST*, 10, 2008.
19. S. Neema, T. Bapty, J. Gray, and A. Gokhale. Generators for Synthesis of QoS Adaptation in Distributed Real-time Embedded Systems. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, pages 236–251, Pittsburgh, PA, Oct. 2002.
20. Object Management Group. *Lightweight CCM RFP*, realtime/02-11-27 edition, Nov. 2002.
21. Object Management Group. *Real-time CORBA Specification*, 1.2 edition, Jan. 2005.
22. T. Ritter, M. Born, T. Unterschütz, and T. Weis. A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In *Proceedings of the* 36^{th} *Hawaii International Conference on System Sciences (HICSS'03)*, page 318, Honolulu, HI, Jan. 2003.
23. Robby, M. Dwyer, and J. Hatcliff. Bogor: An Extensible and Highly-Modular Model Checking Framework. In *Proceedings of the* 4^{th} *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, Helsinki, Finland, September 2003. ACM.
24. N. Shankaran, D. C. Schmidt, Y. Chen, X. Koutsoukous, and C. Lu. The Design and Performance of Configurable Component Middleware for End-to-End Adaptation of Distributed Real-time Embedded Systems. In *Proc. of the 10th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2007)*, Santorini Island, Greece, May 2007.
25. D. C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In *Software Product Lines: Experience and Research Directions*, volume 576, pages 353–370, Aug 2000.

26. J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. 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*, pages 58–69, Toronto, Canada, 2003. IEEE Computer Society.
27. L. Zhu, N. B. Bui, Y. Liu, and I. Gorton. MDABench: Customized benchmark generation using MDA. *Journal of Systems and Software*, 80(2):265–282, Feb. 2007.
28. J. A. Zinky, D. E. Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.