DQML: A Modeling Language for Configuring Distributed Publish/Subscribe Quality of Service Policies*

Joe Hoffert, Douglas Schmidt, and Aniruddha Gokhale

Institute for Software Integrated Systems, Dept. of EECS,
Vanderbilt University, Nashville, TN, USA 37203
`{jhoffert,schmidt,gokhale}@dre.vanderbilt.edu`
`http://www.dre.vanderbilt.edu`

Abstract. Many publish/subscribe (pub/sub) middleware platforms provide flexibility in configuring policies that affect end-to-end quality of service (QoS). While the functionality and tunability of pub/sub middleware has increased, so has the complexity of creating semantically compatible QoS policy configurations. This paper makes two contributions to addressing these challenges. First, it describes how a domain-specific modeling language (DSML) can automate the analysis and synthesis of semantically compatible QoS policy configurations. Second, it empirically evaluates how this DSML increases productivity when generating valid QoS policy configurations. Our experimenta results show a 54% reduction in development effort using DQML over manual methods.

Key words: Configuration Modeling Tools, Pub/Sub Middleware, Domain-Specific Modeling Languages, Data Distribution Service, Modeling Metrics

1 Introduction

Emerging trends for publish/subscribe systems. The use of distributed systems based on publish/subscribe (pub/sub) technologies has increased due to the advantages of scale, cost, and performance over single computers [10, 21]. In contrast to distributed object computing middleware (*e.g.*, Java RMI and CORBA) where clients invoke point-to-point methods on distributed objects, pub/sub middleware disseminates data from suppliers to one or more consumers. Examples of pub/sub middleware include Web Services Brokered Notification [18], the Java Message Service (JMS) [20], the CORBA Event Service [13], and the Data Distribution Service (DDS) [15]. These technologies support data propagation throughout a system using an anonymous subscription model that decouples event suppliers and consumers.

Pub/sub middleware is applicable to a broad range of application domains, such as satellite coordination and shipboard computing environments. This middleware provides policies that affect end-to-end system QoS. Common policies include *persistence* (*i.e.*, saving data for current subscribers), *durability* (*i.e.*, sav-

* This work is supported in part by the AFRL/IF Pollux project and NSF TRUST.

ing data for subsequent subscribers), and *grouped data transfer* (*i.e.*, transmitting and receiving a group of data as an atomic unit).

Challenges in configuring pub/sub middleware. While tunable policies enable fine-grained control of system QoS, a number of challenges arise when developing *QoS policy configurations*, which are combinations of QoS properties that affect overall system QoS. For example, each QoS policy may have multiple parameters associated with it, such as the data topic of interest, data filter criteria, and the maximum number of messages to store when transmitting data. Each parameter can also be assigned a range of values (such as the legal set of topics), a range of integers for the maximum number of data messages stored for transmission, or the set of regular expressions used as filtering criteria.

The QoS policies associated with individual suppliers or consumers collectively determine the overall observed QoS of suppliers and consumers. Not all combinations of QoS policies/parameters deliver the required system QoS, however, and many combinations may not be semantically compatible. It is tedious and error-prone to transform a valid QoS policy configuration design manually to its implementation for a middleware platform.

Solution approach → Model-driven QoS policy configuration. We have developed a domain-specific modeling language (DSML) called the *Distributed QoS Modeling Language* (DQML) to address the challenges described above. In particular, DQML helps developers (1) choose valid sets of values for QoS policies in pub/sub middleware, (2) ensure that these QoS policy configurations are semantically compatible, *i.e.*, they do not conflict with each other, and (3) automate the transformation of a QoS policy configuration design into the correct pub/sub middleware-specific implementation.

Our prior work [7] briefly outlined some QoS policy configuration challenges and summarized our DSML-based solution approach. This paper significantly expands our prior work to (1) provide in-depth analysis of the challenges and present additional lessons learned, (2) empirically evaluate the productivity gains of our DSML solution, and (3) compare our work with related efforts.

2 Motivating Example: NASA's Magnetospheric Multiscale Mission

We chose NASA's *Magnetospheric Multiscale* (MMS) Mission [19] as a case study to showcase the complexities of configuring QoS policies in pub/sub middleware. MMS comprises five co-orbiting and coordinated satellites instrumented identically to study various aspects of the earth's magnetosphere, *e.g.*, turbulence in key boundary regions, magnetic reconnection, and charged particle acceleration. The satellites can be (re)positioned into different temporal/spatial relationships, *e.g.*, to construct a three dimensional view of the field, current, and plasma structures.

An example MMS spacecraft deployment is shown in Figure 1. This deployment includes a non-MMS satellite that communicates with the MMS satellites, as well as a ground station that communicates with the satellites during a high-capacity orbit window. The figure also shows the flow of data between systems involved in the deployment, along with the QoS requirements applicable to the MMS mission.

To transport telemetry data, the MMS satellites are equipped with both downlink and uplink capability. To enable precise coordination for particular types of

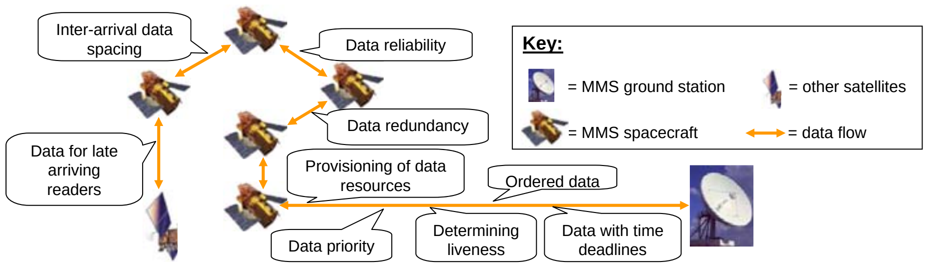


Fig. 1. Example MMS Mission Scenario with QoS Requirements

telemetry and positioning data each satellite gathers, stores, and transmits information regarding neighboring spacecraft. Instrumentation on each satellite is expected to generate ∼250 megabytes of data per day. To enable the satellites to wait for high-rate transmission windows and thereby minimize ground station cost, each satellite also stores up to 2 weeks worth (*i.e.*, 3.5 GB) of data. To meet these data requirements, the pub/sub middleware used for MMS needs to support the QoS policies summarized in Table 1.

MMS Requirement	**Description**
Redundancy	data redundancy (store data on another satellite)
Durability	making data available at a later time for analysis
Presentation	maintain message ordering and granularity
Transport priority	prioritizing data transmissions
Time-based filtering	flow control to handle slow consumers
Deadline	deadlines on receipt of data
Reliability	no loss of critical data
Resource limits	effective provisioning of resources
Liveliness	assurances of properties when spacecraft is unavailable

Table 1. MMS pub-sub QoS Policy Requirements

A challenge for MMS developers is to determine how the interaction of the QoS policies listed in Table 1 impacts the deployed system. Key issues to address involve detecting conflicting QoS settings and ensuring proper behavior of the system in light of such conflicts. Not all combinations of QoS policies and parameter values are semantically compatible, *i.e.*, only a subset actually make sense and provide the needed capabilities. Ideally, incompatibilities in QoS policy configurations should be detected *before* the MMS system runs so modifications will be less costly and easier to implement, validate, and optimize.

3 Analyzing QoS Policy Configuration Challenges and Solutions

This section explores the challenges of generating QoS policy configurations for pub/sub middleware and presents DSML-based solutions. We analyze these challenges in the context of a prototype MMS mission (see Section 2) implemented using OMG Data Distribution Service (DDS) [15] pub/sub middleware (see Section 3.1). We selected DDS as our middleware platform due to its extensive support for QoS policies that are relevant to the MMS mission case study.

3.1 Overview of the OMG Data Distribution Service (DDS)

The OMG DDS specification defines a standard architecture for exchanging data in pub/sub systems. DDS provides a global data store in which publishers and subscribers write and read data, respectively. DDS provides flexibility and modular structure by decoupling: (1) *location*, via anonymous publish/subscribe, (2) *redundancy*, by allowing any numbers of readers and writers, (3) *time*, by providing asynchronous, time-independent data distribution, and (4) *platform*, by supporting a platform-independent model that can be mapped to different platform-specific models, such as C++ running on VxWorks or Java running on Real-time Linux.

The DCPS entities in DDS include *topics*, which describe the type of data to be written or read, *data readers*, which subscribe to the values or instances of particular topics, and *data writers*, which publish values or instances for particular topics. Properties of these entities can be configured using combinations of the DDS QoS policies shown in Table 2. Moreover, *publishers* manage groups of data

DDS QoS Policy	Description
Deadline	Determines rate at which periodic data is refreshed
Destination Order	Sets whether data sender or receiver determines order
Durability	Determines if data outlives the time when written or read
Durability Service	Details how durable data is stored
Entity Factory	Sets enabling of DDS entities when created
Group Data	Attaches application data to publishers, subscribers
History	Sets how much data is kept to be read
Latency Budget	Sets guidelines for acceptable end-to-end delays
Lifespan	Sets time bound for “stale” data
Liveliness	Sets liveness properties of topics, data readers, data writers
Ownership	Controls writer(s) of data
Ownership Strength	Sets ownership of data
Partition	Controls logical partition of data dissemination
Presentation	Delivers data as group and/or in order
Reader Data Lifecycle	Controls data and data reader lifecycles
Reliability	Controls reliability of data transmission
Resource Limits	Controls resources used to meet requirements
Time Based Filter	Mediates exchanges between slow consumers and fast producers
Topic Data	Attaches application data to topics
Transport Priority	Sets priority of data transport
User Data	Attaches application data to DDS entities
Writer Data Lifecycle	Controls data and data writer lifecycles

Table 2. DDS QoS Policies

writers and *subscribers* manage groups of data readers.

Each QoS policy has ∼2 parameters, with the bulk of the parameters having a large number of possible values, *e.g.*, a parameter of type long or character string. Section 3.2 shows that not all QoS policies are applicable to all DCPS entities nor are all combinations of policy values semantically compatible.

3.2 DDS QoS Policy Configuration: Challenges and DSML-based Solutions

In the context of DDS and the MMS case study, we developed a DSML-based solution to four types of challenges that arise when creating QoS policy configurations.[2] We chose a DSML-based solution over other common solution techniques (such as manually-implementing point- and pattern-based [11] solutions) since DSMLs can ensure (1) proper semantics for specifying QoS policies and (2) all parameters for a particular QoS policy are properly specified and used correctly, as described in Section 1. DSMLs can also detect many types of QoS policy configuration problems *at design time* and can automatically generate implementation artifacts (*e.g.*, source code and configuration files) that reflect design intent.

Challenge 1: Managing QoS Policy Configuration Variability.

Context. DDS provides three points of variability with respect to QoS policy configurations: (1) the associations between a single DDS entity and two or more QoS policies, (2) the associations between two or more entities, and (3) the number and types of parameters per QoS policy.

Problem. When creating a DDS QoS policy configuration, associations are made between various entities *e.g.*, between a data writer sending collected data from an MMS satellite and the publisher that manages the data writer. Not all possible associations are valid, however. For example, the association between a data writer and a subscriber is invalid since a subscriber manages one or more data readers and not data writers. If the rules governing valid associations between entities are not obeyed when associations are created the QoS policy configuration will be invalid.

Associations can be made not only between DDS entities but also between a DDS entity and the QoS policies. Not all QoS policies are valid for all DDS entities, however. For instance, associating a Presentation QoS policy with an MMS ground station's data reader is invalid. The rules that determine which QoS policies can be associated with which DDS entities must be considered when creating valid QoS policy configurations.

Finally, the number and types of parameters differ for each QoS policy type. The number of parameters for any one QoS policy ranges from one (*e.g.*, Deadline QoS Policy) to six (*e.g.*, Durability Service QoS Policy). The parameter types for any one QoS policy also differ. The parameter types include boolean, string, long, struct, and seven different types of enums. It is hard to track the number of parameters a particular QoS policy has manually; it is even harder to track the valid range of values that any one single parameter can have.

General DSML-based solution approach. A DSML can ensure that only appropriate associations are made between entities and QoS policies. In addition, a DSML can list the parameters and default values of any selected QoS policy. DSMLs ensure that only valid values are assigned to the QoS policy parameters. For example, a DSML can raise an error condition if a string is assigned to a parameter of type long. Section 4.2 describes how DQML addresses the QoS policy configuration variability challenge by allowing only valid values to be assigned to parameters and checking for valid associations between QoS policies and entities.

Challenge 2: Ensuring QoS compatibility.

Context. DDS defines constraints for compatible QoS policies. Table 3 lists the QoS policies that can be incompatible and the relevant types of entities for those

[2] While DDS's rich set of QoS policies makes it a particularly relevant platform, similar analysis and solutions are also applicable to other pub/sub middleware and application domains.

policies. Incompatibility applies to QoS policies of the same type, *e.g.*, *reliability*, across multiple types of entities, *e.g.*, *data reader* and *data writer*.

QoS Policies	Affected DDS Entities
Deadline	Topic, data reader, data writer
Destination Order	Topic, data reader, data writer
Durability	Topic, data reader, data writer
Latency Budget	Topic, data reader, data writer
Liveliness	Topic, data reader, data writer
Ownership	Topic, data reader, data writer
Presentation	Publisher, subscriber
Reliability	Topic, data reader, data writer

Table 3. Potential Incompatible DDS QoS Policies

Problem. When compatibility constraints are violated, data will not flow between DDS data writers and data readers, *i.e.*, compatibility impacts topic dissemination. For example, an incompatibility between reliability QoS policies will occur if an MMS ground station requests reliable data readers, but an MMS spacecraft only offers best-effort data writers. The data will not flow between the spacecraft and the ground station because the values of the QoS policies are incompatible, as shown in Figure 2.

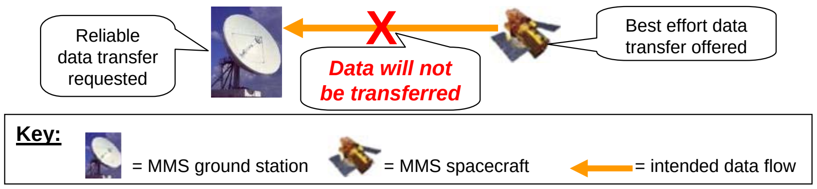

Reliable data transfer requested
X
Data will not be transferred
Best effort data transfer offered
Key:
= MMS ground station
= MMS spacecraft
= intended data flow

Fig. 2. Incompatible MMS Ground Station and Spacecraft Reliability QoS

General DSML-based solution approach. A DSML can include compatibility checking in the modeling language itself. A DSML user can invoke compatibility checking to make sure that the QoS policy configuration specified is valid. If incompatible QoS policies are detected the user is notified *at design time* and given details of the incompatibility. Section 4.2 describes how DQML addresses the QoS compatibility challenge by providing compatibility constraint checking on QoS policy configurations.

Challenge 3: Ensuring QoS consistency.

Context. The DDS specification defines when QoS policies are inconsistent, *i.e.*, when multiple QoS policies associated with a single DCPS entity are not valid. Table 4 describes the consistency constraints for QoS policies associated with a single DDS entity. For example, an inconsistency between the *Deadline* and

Consistency Constraints for QoS Policies
Deadline.period $\geq$ Time_Based_Filter.minimum separation
Resource_Limits.max_samples $\geq$ Resource_Limits.max_samples_per_instance
Resource_Limits.max_samples_per_instance $\geq$ History.depth

Table 4. DDS QoS Consistency Constraints

Time-based Filter QoS policies occurs if an MMS ground station tries to set the *Deadline* QoS policy's deadline period to 5 ms and the *Time-based Filter* QoS policy's minimum separation between incoming pieces of data to 10 ms, as shown in Figure 3. This invalid configuration violates the DDS constraint of deadline period $\geq$ minimum separation.

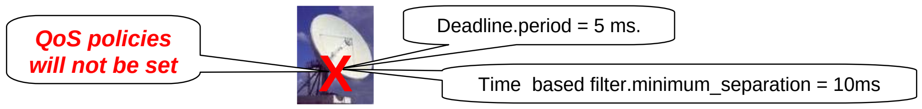

QoS policies will not be set
Deadline.period = 5 ms.
Time based filter.minimum_separation = 10ms

Fig. 3. Inconsistent QoS Policies for an MMS Ground Station

Problem. Manually checking for all possible consistency constraint violations is tedious and error-prone for non-trivial pub/sub systems.

General DSML-based solution approach. A DSML can include consistency checking in the modeling language itself. As with compatibility checking, DSML users can invoke consistency checking to ensure that the QoS policy configuration is valid. If inconsistent QoS policies are found, users are notified *at design time* with detailed information to help correct the problem. Section 4.2 describes how DQML addresses the QoS consistency challenge by providing consistency constraint checking on QoS policy configurations.

Challenge 4: Ensuring Correct QoS transformation.

Context. After a valid QoS policy configuration has been created it must be correctly transformed from design to implementation.

Problem. A conventional approach is to (1) document the desired QoS policies, parameters, values, and associated entities often in an *ad hoc* manner (*e.g.*, using handwritten notes or conversations between developers) and then (2) transcribe this information into the source code. This *ad hoc* process creates opportunities for accidental complexities, however, since the QoS policies, parameters, values, and related entities can be misread, mistyped, or misunderstood. The QoS policy configurations encoded in the system may therefore differ from the valid configurations intended originally.

General DSML-based solution approach. A DSML can provide model interpreters to generate correct-by-construction[3] implementation artifacts. The interpreters iterate over the QoS policy configuration model designed in the DSML to create appropriate implementation artifacts (*e.g.*, source code, configuration files) that will correctly recreate the QoS policy configuration as designed. Section 4.2 describes how DQML addresses the challenge of correct QoS transformation by providing an interpreter that traverses the model and generates implementation specific artifacts.

4 The Distributed QoS Modeling Language (DQML)

The *Distributed QoS Modeling Language* (DQML) is a DSML that automates the analysis and synthesis of semantically compatible DDS QoS policy configurations. We developed DQML using the Generic Modeling Environment (GME) [1], which is a meta-programmable environment for creating DSMLs. This section describes the structure and functionality of DQML and explains how it resolves the challenges from Section 3.2 in the context of DDS and the MMS case study.

4.1 Structure of the DQML Metamodel

The DQML metamodel constrains the possible set of models for QoS policy configurations, as described below.

[3] In this paper "correct-by-construction" refers to QoS policy configuration artifacts that faithfully transfer design configurations into implementation and deployment.

Scope. The DQML metamodel includes all DDS QoS policy types shown in Table 2, but supports only DDS entity types that have QoS policies associated with them. In addition to topics, data readers, and data writers previously mentioned, DQML can associate QoS policies with (1) *publishers*, which manage one or more data writers, (2) *subscribers*, which manage one or more data readers, (3) *domain participants*, which are factories for DDS entities for a particular domain or logical network, and (4) *domain participant factories*, which generate domain participants. While other entities and constructs exist in DDS, none directly use QoS policies and are thus excluded from DQML.

Figure 4 shows a portion of the DQML metamodel relevant to the Deadline QoS policy and its relationships to applicable DDS entities, *e.g.*, data reader, data writer, and topic. Figure 5 shows a portion of the DQML metamodel relevant to the OCL constraint placed on the Deadline QoS policy to ensure semantic compatibility. The compatibility constraints are associated with a topic since compatibility between a data reader and a data writer is determined by a common topic. This

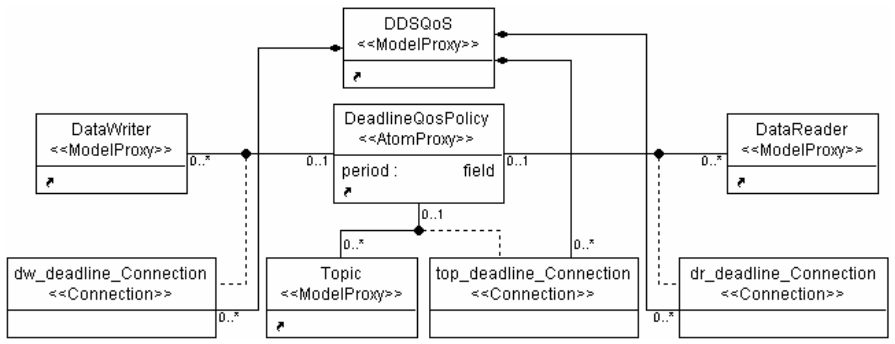


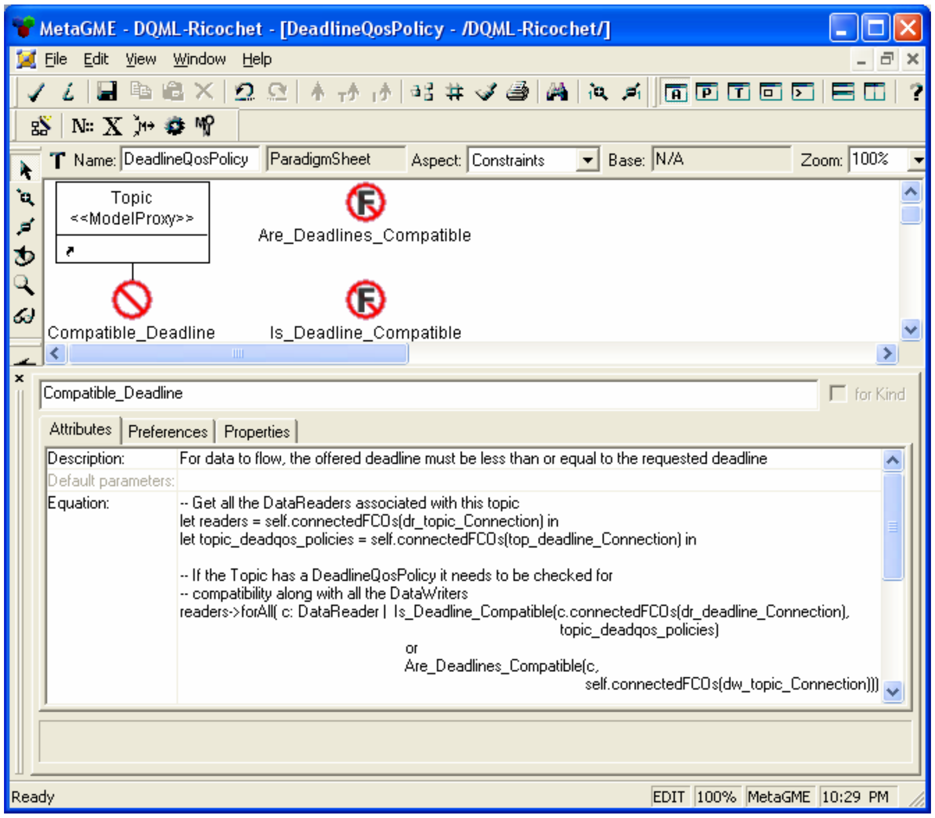
Fig. 4. Deadline QoS Policy Relationships (UML notation)

figure shows the appropriate relationships and the number of associations. Other QoS policies are modeled in this way, along with the parameters and constraints for each policy.

Associations between entities and QoS policies. DQML supports associations between DDS entities and QoS policies rather than having DDS entities contain or own QoS policies. This metamodel design decision allows greater flexibility and ease of constraint error resolution. If QoS policies had been contained by the DDS entities then multiple DDS entities could not share a common QoS policy. Instead, the policy would be manually copied and pasted from one entity to another, thereby incurring accidental complexity when designing a QoS policy configuration.

In contrast, DQML supports multiple DDS entities having the same QoS policy by allowing modelers to create a single QoS policy with the appropriate values. Modelers can then create associations between the applicable DDS entities and the QoS policy. This approach also simplifies constraint errors resolution, *e.g.*, if constraint errors are found, the offending entities can be associated with a common QoS policy to eliminate the compatibility error.

Constraint definition. The DDS specification defines constraints placed on QoS policies for compatibility and consistency. The DQML metamodel uses



Description: For data to flow, the offered deadline must be less than or equal to the requested deadline

Default parameters:

Equation:

```
-- Get all the DataReaders associated with this topic
let readers = self.connectedFCOs(dr_topic_Connection) in
let topic_deadqos_policies = self.connectedFCOs(top_deadline_Connection) in

-- If the Topic has a DeadlineQosPolicy it needs to be checked for
-- compatibility along with all the DataWriters
readers->forAll( c: DataReader |  Is_Deadline_Compatible(c.connectedFCOs(dr_deadline_Connection),
                                                     topic_deadqos_policies)
                                  or
                                  Are_Deadlines_Compatible(c,
                                                         self.connectedFCOs(dw_topic_Connection)))
```

Fig. 5. Deadline QoS Policy Compatibility Constraints

GME's Object Constraint Language (OCL) [22] implementation to define these constraints. As noted in Section 3.2 for challenges 2 and 3, compatibility constraints involve a single type of QoS policy associated with more than one DDS entity, whereas consistency constraints involve a single DDS entity with more than one QoS policy. Both types of constraints are defined in the metamodel and can be checked when explicitly initiated by users.

To maximize flexibility, DQML does not enforce semantic compatibility constraints automatically in the metamodel since users may only want to model some parts of a DDS application, rather than model all required entities and QoS policies. Only checking constraints when initiated by modelers enables this flexibility. Conversely, association constraints (*i.e.*, the valid associations between DDS entities and QoS policies) *are* defined in the metamodel and are thus checked automatically when associations are specified.

4.2 Functionality of DQML

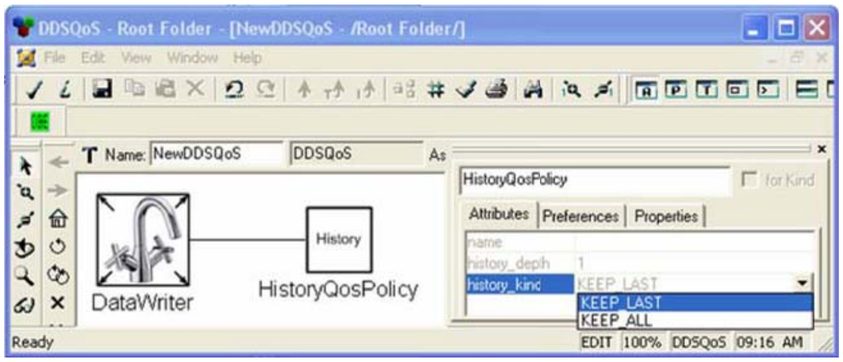
DQML allows developers to designate any number of DDS entity instances involved with QoS policy configuration. For example, DQML supports seven DDS entity types that can be associated with QoS policies, as shown in Figure 6. QoS policies can be created and associated with these entities as described below.

Fig. 6. DDS Entities Supported in DQML

Specification of QoS policies. DQML allows developers to designate the DDS QoS policies involved with a QoS policy configuration. DQML supports all DDS policies, along with their parameters, the appropriate ranges of values, and the default parameter values. Developers can then change default settings for QoS policy parameters as needed. Moreover, if a QoS policy parameter has a limited range of values, DQML enumerates only these specific values and ensures that only one of these values is assigned to the parameter.

DQML also ensures that the type of value assigned is appropriate. For example it ensures that a character value is not assigned to a parameter that requires an integer value. The DQML interpreter externalizes the parameter values (whether set explicitly or by default) so that no QoS policy has uninitialized parameters.

Figure 7 shows an example of how DQML addresses the challenge of managing QoS policy configuration variability as outlined in Section 3.2. In this example DQML displays the parameters for the history QoS policy along with the default values for the parameters in grey, *i.e.*, `history_depth = 1` and `history_kind = KEEP_LAST`. Since `history_kind` is an enumerated type, DQML lists the valid values when the user selects the parameter. Only one of the valid values can be assigned to the parameter.



name
history_depth 1
history_kind KEEP_LAST
KEEP_LAST
KEEP_ALL

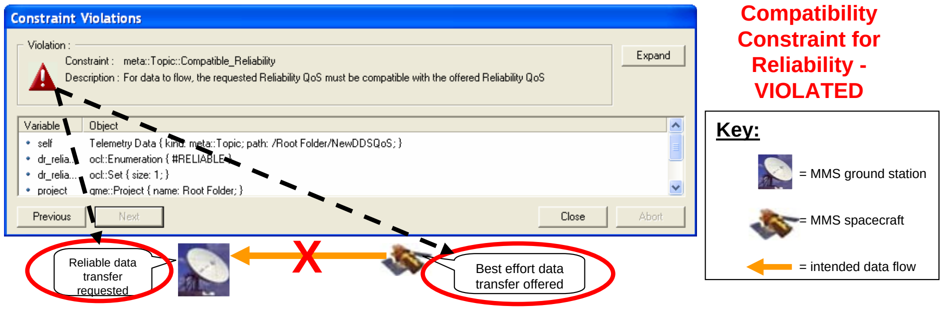
Fig. 7. Example of DQML QoS Policy Variability Management

Association between entities and QoS policies. DQML supports generating associations between the DDS entities themselves and between a DDS entity and the QoS policies. DQML ensures that only valid associations are created *i.e.*, where it is valid to associate two particular types of entities or associate a particular DDS entity with a particular type of QoS policy. DQML will notify developers if the association is invalid and disallow the association at design-time.

Checking compatibility and consistency constraints. DQML supports checking for compatible and consistent QoS policy configurations. Users initiate this checking and DQML reports any violations. Constraint checking in DQML uses default QoS parameter values to determine QoS compatibility and consistency if no values are specified. Developers of QoS policy configurations might explicitly associate only a single QoS policy to an entity and assume no checking for compatibility or consistency is applicable. A constraint violation may exist, however, depending on the interaction of the explicit parameter values and the default values for other entities.

For instance, if developers specify only a single presentation QoS policy in a configuration, associate it with a single subscriber entity, and change the default *access scope* value from *instance* to *topic* or *group*, they may assume no constraint violations occur. The explicit access scope value set on the subscriber is incompatible, however, with the implicit (default) value of *instance* for *any* publisher associated via a common topic.

The constraint resolution problem is further exacerbated by QoS policies that can be associated with a topic entity and then act as the default QoS policy for data readers or writers. For example, the reliability QoS policy can be associated with a data reader, a data writer, or a topic. If the policy is associated with a topic, any data readers or data writers not explicitly associated with a reliability policy will use the topic's reliability QoS policy. DQML can check this type of QoS association for compatibility and consistency.

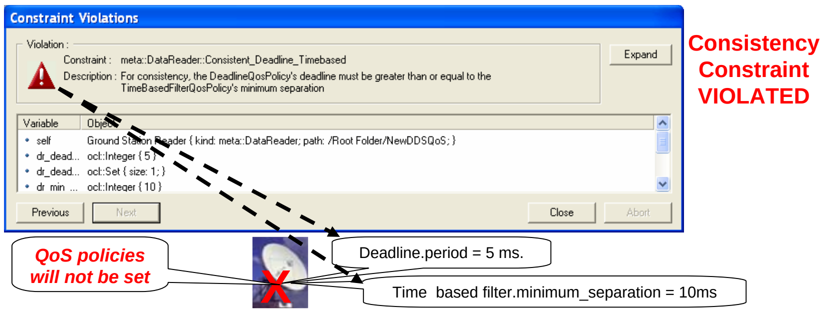


Constraint : meta::Topic::Compatible_Reliability
Description : For data to flow, the requested Reliability QoS must be compatible with the offered Reliability QoS

Variable	Object
self	Telemetry Data { kind: meta::Topic; path: /Root Folder/NewDDSQoS; }
dr_relia...	ocl::Enumeration { #RELIABLE }
dr_relia...	ocl::Set { size: 1; }
project	gme::Project { name: Root Folder; }

Fig. 8. Example of DQML QoS Policy Compatibility Constraint Checking

Figures 8 and 9 show examples of how DQML addresses the challenges of ensuring QoS compatibility and consistency, respectively, as described in Section 3.2. Figure 8 shows how DQML detects and notifies users of incompatible reliability


Constraint Violations
Violation :
Expand
Previous
Next
Close
Abort
Consistency Constraint VIOLATED
QoS policies will not be set
Deadline.period = 5 ms.
Time based filter.minimum_separation = 10ms

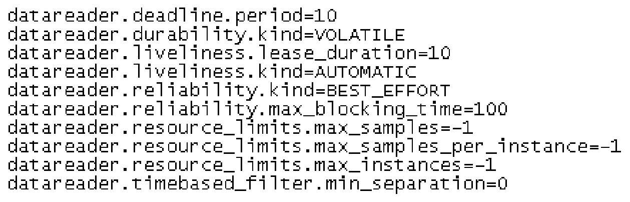
Constraint : meta::DataReader::Consistent_Deadline_Timebased
Description : For consistency, the DeadlineQosPolicy's deadline must be greater than or equal to the TimeBasedFilterQosPolicy's minimum separation

Variable	Object
self	Ground Station Reader { kind: meta::DataReader; path: /Root Folder/NewDDSQoS; }
dr_dead...	ocl::Integer { 5 }
dr_dead...	ocl::Set { size: 1; }
dr_min ...	ocl::Integer { 10 }

Fig. 9. Example of DQML QoS Policy Consistency Constraint Checking

QoS policies. Likewise, Figure 9 shows an incompatible deadline period, *i.e.*, 10 is less than the time based filter's minimum separation of 15. Both policies are associated with the same MMS Ground Station data reader. DQML checks the consistency of the modeled QoS policies and notifies users of violations.

Transforming QoS policy configurations from design to implementation. Figure 10 shows how DQML addresses the challenge of correctly transforming QoS policy configurations from design to implementation, as described in Section 3.2. In this example, DQML generates a QoS policy configuration file for an MMS satellite data reader. QoS policies associated with the data reader along with values for the policies are shown. This file can then be integrated into the MMS implementation to ensure the desired QoS policy configuration.



```
datareader.deadline.period=10
datareader.durability.kind=VOLATILE
datareader.liveliness.lease_duration=10
datareader.liveliness.kind=AUTOMATIC
datareader.reliability.kind=BEST_EFFORT
datareader.reliability.max_blocking_time=100
datareader.resource_limits.max_samples=-1
datareader.resource_limits.max_samples_per_instance=-1
datareader.resource_limits.max_instances=-1
datareader.timebased_filter.min_separation=0
```

Fig. 10. Example QoS Policy Configuration File

5 DQML Productivity Analysis for the MMS Case Study

In this section we analyze the pros and cons of DQML by applying it in the context of the *DDS Benchmarking Environment* (DBE) to evaluate the QoS behavior of the MMS scenario presented in Figure 1. DBE is a suite of software tools that can examine and evaluate various DDS implementations [8]. DBE requires correct QoS policy settings so that data will flow as expected. If these policy settings are semantically incompatible QoS evaluations will not run properly. DBE uses a set of Perl scripts that launches executables for the DDS application, *e.g.*, to deploy data readers and data writers onto specified nodes. For each data reader and data writer DBE also deploys a QoS policy settings file that is currently generated manually.

This section presents the results of a productivity analysis using DQML. In particular, we present the productivity benefit and the break-even point of using DQML vs. manually implementing QoS policy configurations for DBE. Manual implementation of configurations is applicable to both the point- and pattern-based solutions presented in previous work [7] since neither approach provides implementation guidance.

5.1 The DQML DBE Interpreter

To support DBE and its need to generate correct QoS policy configurations we developed a DQML interpreter that generates QoS policy parameter settings files for the data readers and data writers that DBE configures and deploys. This interpreter can also accommodate other DCPS entities, *e.g.*, topics, publishers, and subscribers. All QoS policies from a DQML model are output for the data readers and data writers.

The DQML interpreter creates one QoS policy parameter settings file for each data reader or data writer that is modeled. The names of the files are generated by using the name of the data reader or data writer prepended with either "DR" or "DW" plus the current count of data readers or data writers processed (*e.g.*, `DR1_Satellite1.txt`). The filename prefix is generated to ensure that a unique filename is created since the names of the data readers and data writers modeled in DQML need not be unique.

A common DBE use-case for DQML thus becomes (1) model the desired DCPS entities and QoS policies in DQML, (2) invoke the DBE interpreter to generate the appropriate QoS settings files, and (3) execute DBE to deploy data readers and data writers using the generated QoS settings files.

5.2 Productivity Analysis

Scope. DBE currently deals only with DDS data readers and data writers. Our productivity analysis therefore focuses on the QoS parameters relevant to data readers and data writers. (Similar analysis can be done for other types of DDS entities associated with QoS policies.) At a minimum, in the MMS scenario each MMS satellite, non-MMS satellite, and ground station will have a data writer and data reader to send and receive data, respectively, which yields seven data readers and seven data writers to configure. This scenario provides the minimal baseline since production satellites and ground stations typically have many data writers and data readers for use in sending and receiving not only to other systems but also for use internally between various subsystems.

QoS Policy	**# of Params**	**Param Type(s)**
Deadline	1	int
Destination Order	1	enum
Durability	1	enum
Durability Service	6	5 ints, 1 enum
History	2	1 enum, 1 int
Latency budget	1	int
Lifespan	1	int
Liveliness	2	1 enum, 1 int
Ownership	1	enum
Ownership Strength	1	int
Reliability	2	1 enum, 1 int
Resource Limits	3	3 ints
Transport Priority	1	int
User Data	1	string
Writer Data Lifecycle	1	bool
Total Parameters	**25**	

Table 5. DDS QoS Policies for data writers

QoS Policy	**# of Params**	**Param Type(s)**
Deadline	1	int
Destination Order	1	enum
Durability	1	enum
History	2	1 enum, 1 int
Latency budget	1	int
Liveliness	2	1 enum, 1 int
Ownership	1	enum
Reader Data Lifecycle	1	int
Reliability	2	1 enum, 1 int
Resource Limits	3	3 ints
Time Based Filter	1	int
User Data	1	string
Total Parameters	**17**	

Table 6. DDS QoS Policies for data readers

A data writer can be associated with 15 QoS policies with a total of 25 parameters, as shown in Table 5. A data reader can be associated with 12 QoS policies with a total of 17 parameters, as shown in Table 6. The total number of relevant QoS parameters for DBE is thus $\mathbf{17 + 25 = 42}$. Each QoS parameter value for a data reader or writer corresponds to one line in the QoS policy parameter settings file for DBE, as shown in Figure 10.

Interpreter development. DQML's DBE interpreter was developed using GME's Builder Object Network (BON2) framework, which generates C++ code using the Visitor pattern [4]. Within BON2, developers of the DQML DBE interpreter need only modify and add certain portions to the framework that are called to process the particular DSML. In particular, BON2 provides a C++ visitor class with virtual methods (*e.g.*, visitModelImpl, visitConnectionImpl, visitAtomImpl) that the developer subclasses and then overrides the virtual methods.

In BON2, the DDS entities supported in DQML are referred to as model implementations. The DBE interpreter is thus only concerned with overriding the visitModelImpl method. When the BON2 framework invokes this method it passes as an argument a model implementation. A model implementation has methods to (1) traverse the associations a DDS entity has using the getConnEnds method and specifying the relevant QoS policy association as an input parameter (*e.g.*, the association between a data reader and a reliability QoS policy), (2) retrieve the connected QoS policy, and (3) obtain the attributes of the associated QoS policy using the policy's getAttributes method.

The DQML-specific code for the DBE interpreter contains ∼160 C++ statements that were developed specifically for DQML and DBE. The C++ development effort for the DBE interpreter need only occur once. In particular, no QoS policy configuration for DBE incurs this development overhead since the interpreter already exists. The development effort is included *only* for comparison with manually implemented QoS policy configurations.

The interpreter code is fairly straightforward once developers understand how to navigate the model in the BON2 framework and access the appropriate information. Although developers should be familiar with the Visitor pattern [4] (since the BON2 framework uses it heavily), they only need define the appropriate methods for the automatically generated Visitor subclass. In general, the DQML interpreter code specific to DBE gathers model information, creates the QoS settings files, and outputs the settings into the QoS settings files.

The most challenging part of developing DQML's DBE interpreter is navigating through the model's QoS policy elements and related entities using the BON2 framework. Conversely, the most challenging aspects of handcrafting QoS policy configurations are (1) maintaining a global view of the model to ensure compatibility and consistency and (2) remembering the number of and valid values for the parameters of the various QoS policies. For non-trivial QoS policy configurations, therefore, developing the DQML-specific C++ code for the interpreter is no more complex than manually ensuring that the QoS settings in settings files are valid, consistent, compatible, and correctly represent the designed configuration.

Analysis for the MMS scenario. As a conservative approximation, the creation and use of the DBE interpreter for DQML has its break-even point for a *single* QoS policy configuration when there are at least 160 QoS policy parameter settings needed, which correlates to the 160 C++ statements for DQML's DBE interpreter. As shown in Figure 11, using the results for QoS parameters in Tables 5 and 6 for data readers and data writers, the break-even point equates to ∼10 data readers, ∼7 data writers, or some combination of data readers and data writers where the QoS settings are greater than or equal to 160 (*e.g.*, 5 data readers and 3 data writers = 160 QoS policy parameter settings).

From the analysis above—and using the minimal MMS scenario in Figure 1 of 7 data writers and 7 data readers—the total number of QoS parameters to consider is 7 * 25 (for data writers) + 7 * 17 (for data readers) = 294. This number exceeds the 160 lines of C++ code developed for the DBE interpreter and shows that the minimal MMS scenario warrants the use of DQML and the creation and use of the DBE interpreter. Using DQML for this scenario provides a 160 ÷ 294 = 54% reduction in development effort as compared to manual methods.

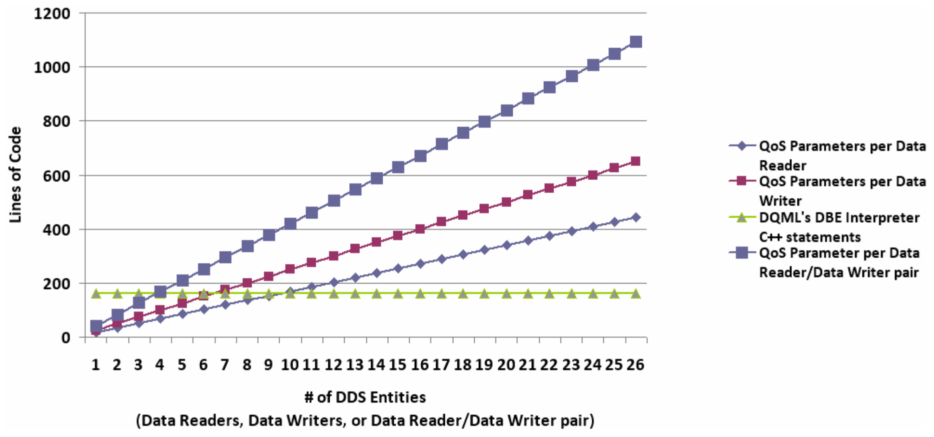


Fig. 11. Metrics for Manual Configuration vs. DQML's Interpreter

Generalized analysis. The break-even analysis above is relevant to generating a single QoS policy configuration. The analysis does not consider any subsequent modifications to an existing configuration or development of new configurations for DBE that would not require any modifications to interpreter code. Changes made to a configuration also require that developers (1) maintain a global view of the model to ensure compatibility and consistency and (2) remember the number of, and valid values for, parameters of the various QoS policies being modified. These challenges still exist when changing an already valid QoS policy configuration.

Moreover, there may be thousands of data readers and writers in large-scale DDS systems, *e.g.*, shipboard computing or air-traffic management environments [3]. Assuming 1,000 data readers and 1,000 data writers, the number of QoS parameters to manage is **17 * 1000 + 25 * 1000 = 42,000**. This number does not include QoS parameter settings for other DDS entities such as publishers, subscribers, and topics. For such large-scale DDS systems the development cost of the DQML interpreter in terms of lines of code is amortized by more than 200 times (*i.e.*, 42,000 / 160 = 262.5).

6 Related Work

This section compares our work on DQML with related R&D efforts.

DSMLs for configuring QoS. There are currently several DSMLs developed to model QoS requirements for distributed real-time embedded (DRE) systems. The *Distributed QoS Modeling Environment* (DQME) [6] is a modeling tool that targets essential elements of dynamic QoS adaptation. DQME is a hybrid of domain-specific modeling and run-time QoS adaption methods, with emphasis on adapting QoS to changing conditions with limited system resources. DQME focuses on QoS solution exploration of a running system by providing run-time QoS adaptation strategies as modeling elements to be incorporated into an existing DSML. In contrast, DQML focuses on design-time generation of semantically compatible QoS policy configurations and correct application-specific implementation artifacts for data-centric pub/sub middleware, such as DDS. In this way, DQML ensures that data for a particular application using pub/sub technology will be disseminated as intended.

The *Options Configuration Modeling Language* (OCML) [2] is a DSML that aids developers in setting compatible *component* configuration options for the system being developed whereas DQML models QoS policy configuration for data-centric middleware that can be applicable across *various endpoints* such as processes, objects, or components. OCML is a modeling language intended to be domain-independent that captures complex DRE middleware and application configuration information along with QoS requirements. It currently supports configuration management only for distributed object computing (DOC) architectures rather than data-centric pub/sub architectures like DDS. This difference is important because the endpoints receiving data in a system utilizing DDS do not specify details of the type and implementation characteristics of the end points. For instance, these endpoints could be processes, objects, or components. DQML models QoS policy configuration for data-centric middleware that can be applicable across these various endpoints. Conversely, OCML is focused to aid developers set component configuration options for the system being developed.

Runtime monitoring. Real-time Innovations Inc. (`www.rti.com`) and Prism Technologies (`www.prismtechnologies.com`) have developed DDS products along with MDE tools that monitor and analyze the flow of data within a system using DDS. These tools help verify that a system is functioning as designed for a particular QoS policy configuration and for a particular point of execution. Discovering configuration problems at run-time is very late in the development cycle, when problems are more expensive and obtrusive to fix. Moreover, these tools are designed only for the vendor-specific DDS implementation.

In contrast, DQML allows QoS policy designers to create semantically compatible QoS policies at design time. DQML can also use the QoS policy configuration model to construct semantically compatible and syntactically correct implementation artifacts (*e.g.*, source code, configuration files) that can be incorporated into the system implementation without human intervention. Moreover, DQML is not DDS implementation specific but is more generally applicable.

QoS Profiles. The Unified Modeling Language (UML) [16, 17] provides a profile for modeling QoS properties and mechanisms [14]. The profile specifies a notation for various QoS categories within UML such as throughput, latency, security, and scalability. The profile does not, however, provide explicit support for all the QoS policies modeled in DQML although extensions to the profile can be made to support arbitrary QoS policies. The profile also does not provide automated enforcement of semantic compatibility between QoS properties as is supported by DQML.

7 Concluding Remarks

DQML is a DSML we developed to address key challenges of pub/sub middleware, including (1) managing QoS policy configuration variability, (2) developing semantically compatible configurations, and (3) correctly transforming QoS policy configurations from design to implementation. The following lessons learned summarize our experience using DQML to model QoS policy configurations for the OMG Data Distribution Service (DDS) in the context of the MMS mission.

• **OCL presents a significant learning curve for typical application developers.** Many application developers who are accustomed to using a functional or object-oriented language, such as Java, C, or C++, are not familiar with rule-based constraint languages, such as OCL. Moreover, tool support for OCL is often rudimentary, *e.g.*, limited debugging support, which impedes productivity. In future work we plan to address enforcing constraints by evaluating other constraint solving technologies, such as the Constraint Logic Programming Finite Domain (CLP(FD)) solver [12].

• **DSMLs should build upon pattern knowledge.** A DSML can benefit from the knowledge already documented in configuration patterns by incorporating these patterns into the DSML itself. This approach ensures that different types of patterns provide semantic compatibility and are implemented correctly. We are therefore enhancing DQML to support patterns and higher level services (*e.g.*, security and fault tolerance [9]).

• **Run-time feedback provides crucial system performance insight.** While DQML ensures valid QoS policy configurations, some system properties (*e.g.*, latency and CPU resource utilization) are best evaluated at run-time. Incorporating this type of dynamic information back into a QoS policy configuration model helps increase overall development productivity and system robustness. We are evaluating ways to incorporate runtime and emulation feedback [5] into DQML to enhance QoS policy configuration development.

GME can be downloaded from `www.isis.vanderbilt.edu/Projects/gme`. DQML can be downloaded in GME's XML format along with supporting files from `www.dre.vanderbilt.edu/~jhoffert/DQML/DQML.zip`.

References

1. Akos Ledeczi *et al.* Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.
2. Arvind S. Krishna *et al.* 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, March 2005. IEEE.
3. Douglas C. Schmidt *et al.* Addressing the Challenges of Tactical Information Management in Net-Centric Systems with DDS. *CrossTalk - The Journal of Defense Software Engineering*, March 2008.
4. Erich Gamma *et al. Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
5. James H. Hill *et al.* Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In *Proceedings of the*

12th International Conference on Embedded and Real-Time Computing Systems and Applications, Sydney, Australia, August 2006.

6. Jianming Ye *et al.* A Model-Based Approach to Designing QoS Adaptive Applications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 221–230, Washington, DC, USA, 2004. IEEE Computer Society.
7. Joe Hoffert *et al.* A QoS Policy Configuration Modeling Language for Publish/Subscribe Middleware Platforms. In *Proceedings of International Conference on Distributed Event-Based Systems (DEBS)*, Toronto, Canada, June 2007.
8. Ming Xiong *et al.* Evaluating Technologies for Tactical Information Management in Net-Centric Systems. In *Proceedings of the Defense Transformation and Net-Centric Systems conference*, Orlando, Florida, April 2007.
9. Sumant Tambe *et al.* MoPED: Model-based Provisioning Engine for Dependability, Jun 2008. Submitted to 46th International Conference on Objects, Models, Components, Patterns (TOOLS 2008).
10. Yi Huang and Dennis Gannon. A comparative study of web services-based event notification specifications. *Proceedings of the International Conference on Parallel Processing Workshops*, 0:7–14, 2006.
11. Gordon Hunt. DDS Use Cases: Effective Application of DDS Patterns and QoS. In *OMG's Workshop on Distributed Object Computing for Real-time and Embedded Systems*, Washington, D.C., July 2006. Object Management Group.
12. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
13. Object Management Group. *Event Service Specification Version 1.1*, OMG Document formal/01-03-01 edition, March 2001.
14. Object Management Group. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, v1.0*, OMG Document formal/06-05-02 edition, May 2006.
15. Object Management Group. *Data Distribution Service for Real-time Systems Specification*, 1.2 edition, January 2007.
16. Object Management Group. *Unified Modeling Language Infrastructure, v2.1.2*, OMG Document formal/2007-11-04 edition, November 2007.
17. Object Management Group. *Unified Modeling Language Superstructure, v2.1.2*, OMG Document formal/2007-11-02 edition, November 2007.
18. Organization for the Advancement of Structured Information Standards. *Web Services Brokered Notification Version 1.3*, OASIS Document wsn-ws_brokered_notification-1.3-spec-os edition, October 2006.
19. Surjalal Sharma and Steven Curtis. *Magnetospheric Multiscale Mission*. Springer Verlag, 2005.
20. SUN. Java Messaging Service Specification. `java.sun.com/products/jms/`, 2002.
21. Sasu Tarkoma and Kimmo Raatikainen. State of the Art Review of Distributed Event Systems. Technical Report C0-04, University of Helsinki, 2006.
22. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.