A QoS Policy Configuration Modeling Language for Publish/Subscribe Middleware Platforms*

Joe Hoffert, Douglas Schmidt, and Aniruddha Gokhale
Institute for Software Integrated Systems, Dept of EECS
Vanderbilt University, Nashville, TN 37203
{jhoffert,schmidt,gokhale}@dre.vanderbilt.edu

ABSTRACT

Publish/subscribe (pub/sub) middleware platforms for event-based distributed systems often provide many configurable policies that affect end-to-end quality of service (QoS). Although the flexibility and functionality of pub/sub middleware platforms has matured, configuring their QoS policies in semantically compatible ways has become more complex. This paper makes two contributions to reducing the complexity of configuring QoS policies for event-based distributed systems. First, it evaluates various approaches for managing complex QoS policy configurations in pub/sub middleware platforms. Second, it describes a domain-specific modeling language (DSML) that automates the analysis and synthesis of semantically compatible QoS policy configurations.

Categories and Subject Descriptors

D.3.2 [**Software**]: Programming Languages—*Language Classifications*

Keywords

Pub/sub Middleware, Event-based Distributed Systems, Domain-Specific Modeling Languages, Data Distribution Service

1. INTRODUCTION

With increasing advantages of cost, performance, and scale over single computers, the proliferation of distributed systems in general and distributed event-based systems in particular have increased dramatically in recent years [4]. In contrast to distributed object computing middleware (such as CORBA and Java RMI)—where clients invoke point-to-point methods on distributed objects—pub/sub middleware platforms distribute data from suppliers to (potentially multiple) consumers. Examples of standardized pub/sub middleware include the Java Message Service (JMS) [11], Web Services Notification [9], CORBA Event Service [7], and OMG Data Distribution Service (DDS) [8]. These event-based services allow the propagation of data throughout a system using an anonymous publish/subscribe (pub/sub) model that decouples event suppliers from event consumers.

To support the requirements of a broad spectrum of application domains, pub/sub middleware for event-based distributed systems typically provides many policies that affect end-to-end system QoS properties. Examples of these policies include *persistence*, *i.e.*, determining how much data to save for current subscribers; *durability*, *i.e.*, determining whether to save data for late joining subscribers; and *grouped data transfer*, *i.e.*, determining if a group of data needs to be transmitted and received as a unit.

Each QoS policy may have multiple attributes associated with it, such as the data topic of interest, data filter criteria, and the maximum number of data messages to store when transmitting data. Moreover, each attribute can be assigned one of 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 criteria used for filtering. The research challenges addressed in this paper thus focus on choosing the right set of values for QoS policies and ensuring that these QoS policies are configured together in a semantically compatible way, *i.e.*, that they do not conflict with or contradict each other.

This paper uses DDS as a case study to illustrate the challenges of ensuring semantically compatible QoS policy configurations. DDS is an OMG specification [8] that defines a standard anonymous pub/sub architecture for exchanging data in event-based distributed systems. It provides a global data store in which publishers and subscribers write and read data, respectively. The DDS architecture consists of two layers: (1) the *data-centric pub/sub* (DCPS) layer that provides APIs to exchange topic data based on specified QoS policies and (2) the *data local reconstruction layer* (DLRL) that makes topic data appear local.

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. Various properties of these entities can be configured using combinations of the 22 QoS policies. Moreover, *Publishers* manage groups of data writers and *Subscribers* manage groups of data readers.

Table 1 summarizes all the DDS QoS policies. Each QoS policy has ∼2 attributes with the majority of the attributes having a large number of possible values, *e.g.*, an attribute

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

DEBS '07, June 20-22, 2007, Toronto, Ontario, Canada

of type long or character string. Moreover, not all QoS policies are applicable to all DCPS entities, nor are all combinations of policy values semantically compatible, as described in Sections 3.1 and 4.

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

Table 1: DDS QoS Policies

Semantic compatibility in DDS is accomplished when the combination and interaction of the specified QoS policies produce the overall desired QoS for the system, *i.e.*, when the system executes with the QoS that is intended. It is hard to achieve semantic compatibility using conventional programming techniques, *i.e.*, manually specifying valid QoS policy configurations with particular QoS policies tuned using appropriate values for the attributes of the policies. Achieving semantic compatibility is particularly hard when QoS policies can conflict with each other, *e.g.*, specifying unreliable data transmission for the data sender with reliable data reception for the data receiver. Moreover, this problem is exacerbated in DDS-based distributed systems where the overall QoS policy configuration is not globally defined or controlled but is defined by the aggregation of the locally defined QoS policy configurations.

A promising way to address key QoS policy configuration concerns for event-based distributed systems involves the use of *model-driven engineering* (MDE) tools [10] based on *domain-specific modeling languages* (DSMLs)[12]. The type systems of DSMLs formalize the application structure, behavior, and requirements within particular domains, such as software-defined radios, avionics mission computing, online financial services, warehouse management, or even the domain of pub/sub middleware platforms. DSMLs are described using metamodels, which define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. Developers use DSMLs to build applications using elements of the type system captured by metamodels and express design intent declaratively rather than imperatively.

This paper describes how a particular MDE-based DSML, the *DDS QoS Modeling Language* (DQML), can help alleviate the complexities of QoS policy configuration for event-based distributed systems. QoS policy configurations can be defined as complex when, in general, a human being cannot manually manage the number of QoS policies, policy attributes, possible values for the attributes, and the interactions of the policies to ensure that the configurations are valid. DQML can be used to model QoS policy configurations within the domain context, *i.e.*, using terminology and constructs within the domain of interest, while also checking for valid QoS policy configurations early in development at design-time, when errors require less effort to fix. DQML can also be used to generate correct-by-construction[1] QoS policy configuration artifacts, *e.g.*, QoS policy configuration files that store particular attribute values for QoS policies which the system loads during execution.

The remainder of the paper is organized as follows: Section 2 uses the NASA Magnetospheric Multiscale (MMS) Mission [1] as an example of an event-based distributed system that can benefit from an MDE-based approach to QoS policy configuration; Section 3 highlights the challenges of QoS policy configuration in a DDS-based design for the MMS mission and analyzes various approaches to addressing these challenges; Section 4 describes how we resolve the challenges presented above using DQML; and Section 5 presents lessons learned guiding future work on DQML.

2. MOTIVATING EXAMPLE: NASA'S MAGNETOSPHERIC MULTISCALE MISSION

We chose NASA's *Magnetospheric Multiscale* (MMS) Mission to highlight the challenges of configuring QoS policies in event-based distributed systems. This mission is designed to study particular aspects of the earth's magnetosphere, such as magnetic reconnection, charged particle acceleration, and turbulence, in key boundary regions. MMS utilizes five coorbiting coordinated spacecraft with identical instrumentation to perform the desired measurements. The spacecraft can be (re)positioned into different spatial/temporal relationships so that a three dimensional view of the plasma, field, and current structures can be constructed.

Figure 1 shows an example MMS spacecraft deployment, including a ground station with which the spacecrafts can communicate during a high-capacity window and a non-MMS satellite that can communicate with the MMS spacecraft. The figure also shows data flows between MMS systems and the QoS requirements applicable to the MMS mission outlined below. MMS spacecraft will be equipped with uplink and downlink capability to transport telemetry data. Each spacecraft will need to gather, store, and transmit information regarding neighboring spacecraft to enable precise coordination for particular types of telemetry and positioning. The instruments aboard each spacecraft will also gen-

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

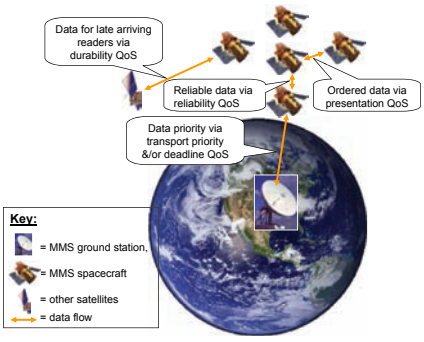

Figure 1: MMS Mission QoS Requirements

erate ~250 megabytes of data per day. To minimize ground station cost each spacecraft will store up to 2 weeks worth (*i.e.,* 3.5 GB) of data so that the spacecraft can wait for high-rate transmission windows. With these data requirements, the DDS pub/sub middleware would be appropriate since its QoS policies support the following capabilities:

a. Storage and later dissemination of data by data writers (such as the spacecraft) to accommodate later data readers (such as other satellites arriving within range). This capability would utilize the DDS durability and durability service QoS policies.
b. Ordered and/or grouped data dissemination so that information transmitted between various MMS systems are received in the same order and the appropriate level of granularity. This capability would utilize the DDS presentation QoS policy.
c. Prioritization of data delivery so that mission-critical or high value data is delivered before lower-valued data. This capability would utilize the DDS transport priority and/or deadline QoS policies.
d. Reliability of data transmission so that no critical data is lost. This capability would utilize the DDS reliability QoS policy.

An MMS mission may need all capabilities listed above, as well as others that occur during the mission. The challenge for MMS developers, however, is to determine what impact the interaction of different QoS policies has on the system, *e.g.*, how will the system behave when the desired QoS settings conflict with each other? As mentioned in Section 1, not all combinations of QoS policy attributes and values are semantically compatible, *i.e.*, only a subset actually make sense and provide the needed capabilities.

For example, the deadline period specified for data reception can conflict with the inter-arrival spacing of data. It is therefore incompatible to specify that (1) data should be received within a particular deadline and (2) the spacing between data reception should at least be greater than that deadline. Likewise, QoS policies between different data publishers and subscribers may conflict, *e.g.*, specifying a publisher's deadline period for sending data to be greater than a subscriber's deadline.

In mission-critical event-based distributed systems, such as MMS, that require QoS support to accomplish system objectives, developers must deal with complex QoS policy configurations. It is necessary to detect incompatibilities between QoS policy configurations, ideally *before* the system begins to run. Different approaches to managing this complexity within the context of the DDS middleware for event-based distributed systems are explored below.

3. QOS POLICY CONFIGURATION CHALLENGES AND ANALYSIS FOR DDS

DDS provides a wide range of QoS capabilities (outlined in Table 1) that can be configured to meet the needs of event-based distributed systems with diverse QoS requirements. DDS's flexible configurability, however, requires careful management of interactions between various QoS policies so that the system behaves as expected. This section uses our MMS example from Section 2 to present the challenges of configuring DDS QoS policies so the system executes as intended.

3.1 DDS QoS Policy Configuration Challenges

The following are three general types of challenges that arise when creating DDS QoS policy configurations.

Challenge 1: QoS compatibility. DDS defines constraints for compatible QoS policies. When these constraints are violated data will not flow between data writers and data readers. For example, an incompatibility between reliability QoS policies will occur if an MMS ground station requests data be sent reliably but an MMS spacecraft offers only a best-effort policy. 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.

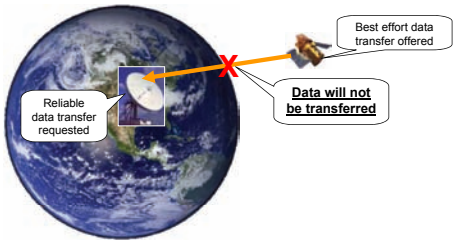

Figure 2: Incompatible MMS Ground Station and Spacecraft Reliability QoS

Table 2 lists the QoS policies that can be incompatible and the relevant types of entities for those policies. Incompatibility applies to QoS policies of the same type, *e.g.*, *Reliability*, across multiple type of entities, *e.g.*, *Data Reader* and *Data Writer*. Section 4.3.1 presents our approach to addressing QoS compatibility challenges.

QoS Policies	Affected DDS Entities
Durability	Topic, Data Reader, Data Writer
Presentation	Publisher, Subscriber
Deadline	Topic, Data Reader, Data Writer
Latency Budget	Topic, Data Reader, Data Writer
Ownership	Topic, Data Reader, Data Writer
Liveliness	Topic, Data Reader, Data Writer
Reliability	Topic, Data Reader, Data Writer
Destination Order	Topic, Data Reader, Data Writer

Table 2: Potential Incompatible DDS QoS Policies

Challenge 2: QoS consistency. DDS also defines when QoS policies are inconsistent indicating that the multiple QoS policies associated with a single DCPS entity are not allowed. For example, an inconsistency between the Deadline and Time-based Filter QoS polices will occur 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.

Figure 3: Inconsistent QoS Policies for an MMS Ground Station

This policy configuration specifies that the ground station requires data at time interval *5 ms* but that it also should only receive data at an interval of at least *10 ms*. Unfortunately, this policy configuration is inconsistent with the DDS constraint that deadline period $\geq$ minimum separation. Table 3 describes consistency constraints for QoS policies associated with a single DDS entity.

Inconsistent QoS Policies
Resource_Limits.max_samples_per_instance < History.depth
Deadline.period < Time_Based_Filter.minimum separation
Resource_Limits.max_samples < Resource_Limits.max_samples_per_instance

Table 3: Consistency of DDS QoS Policies

Even with a relatively small system, the incompatible and inconsistent QoS policies that exist between the reader and writer may not be obvious. This complexity is exacerbated in larger systems, which must consider the number of possible values for each attribute of a QoS policy, the number of attributes for any one QoS policy, the number of QoS policies used, the number of entities associated with QoS policies, the interactions of the QoS policies, and the number of computing nodes in a distributed system.

For example, in the MMS system it is hard to check all incompatible and inconsistent QoS policies manually. If there are any QoS settings that are semantically incompatible, the intended QoS behavior will not be realized and the system will not perform as needed, *e.g.*, data will not be transmitted as required between the spacecraft and the ground stations, between the spacecraft themselves, or within a single spacecraft. Section 4.3.2 describes our approach to addressing QoS consistency challenges.

Challenge 3: Accurate QoS policy configuration transformation. After a valid QoS policy configuration has been designated it must be transformed from design into implementation. A conventional approach to this transformation is to (1) document the desired QoS policies, attributes, values, and associated entities often in an *ad-hoc* manner such as with handwritten notes or conversations between developers, and then (2) transcribe this information into the source code. However, this process creates opportunities for accidental complexities as the QoS policies, attributes, values, and related entities can be misread, mistyped, or misunderstood so that the QoS policy configurations that are encoded in the system differ from the valid configurations originally intended.

3.2 Evaluating Common Alternative Solution Techniques

Below we evaluate three common alternatives for addressing the challenges outlined in Section 3.1 in terms of their ability to document and realize proven QoS policy configurations robustly.

3.2.1 Point Solutions

In this approach, modifications are made to the existing system's QoS policies, feedback is gathered, and further modifications are made based on the feedback. Developers are challenged not only to design a proper QoS policy configuration, but also ensure that the configuration is transformed faithfully from design to implementation.

A point solution approach works best when a configuration expert is available, the configuration is simple, and the configuration need not be maintained nor enhanced. Point solutions, however, make it hard to capture proven QoS policy configurations or leverage from the expertise of others. Moreover, point solutions do not support automated transformation of configuration solutions from design to implementation. Developers must instead rely on human experts who have developed approaches to solving this transformation challenge or must reacquire this expertise.

3.2.2 Patterns-based Solutions

This approach to addressing the DDS QoS policy configuration challenges uses *configuration patterns* for DDS, which document the use of QoS policies that provide management, prioritization, and shaping of a data-flow in a network [5]. For example, system developers could limit access to certain data by utilizing the *DDS Controlled Data Access* pattern, which uses the DDS Partition and User Data QoS Policies along with other DDS elements to provide the desired QoS.

Configuration patterns enable the codification of configuration expertise so it is documented clearly and can be reused broadly. It addresses the problems of the availability of human experts by making the configuration policy expertise generally available. A drawback with a patterns-based approach, however, is that developers are still responsible for implementing the policies manually, which can be tedious and error-prone. Moreover, developers may choose to implement the patterns in different ways, which can impede reuse and integration.

3.2.3 DSML-based Solutions

This approach to addressing the complexity of managing QoS policy configurations involves the use of DSMLs. DSMLs not only codify configuration expertise in the metamodels that are developed for the particular domain but also use an executable form of that expertise to synthesize part or all of an implementation. For example, DSMLs can generate valid QoS configuration files from valid QoS policy configurations modeled in the DSMLs.

DSMLs can also ensure proper semantics for specifying QoS policies and enforce all attributes for a particular QoS policy to be specified and used correctly, as described in Section 1. They can therefore detect many types of QoS configuration problems at design time and can automate the generation of implementation artifacts (*e.g.*, source code and configuration files) that reflect the design intent.

A drawback of using DSMLs is the learning curve required to use them effectively. Moreover, DSMLs typically depend on a particular modeling tool, so users must be familiar with the modeling tool as well as the particular DSML.

4. THE DDS QOS MODELING LANGUAGE

This section describes the *DDS QoS Modeling Language* (DQML), which is a DSML we created using the Generic Modeling Environment (GME)[2] to automate the analysis and synthesis of semantically compatible DDS QoS policy configurations. We summarize the structure and functionality of DQML and then present how DQML helps address the challenges described in Section 3.1.

4.1 Structure of DQML

DDS defines 22 QoS policies shown in Table 1 that control the behavior of DDS applications. DQML models all of these DDS QoS policies, as well as the seven DDS entities that can have QoS policies, *i.e.*, *Data Reader*, *Data Writer*, *Topic*, *Publisher*, *Subscriber*, *Domain Participant*, and *Domain Participant Factory*. Associations between the seven entities themselves and also between the entities and the 22 QoS policies can be modeled taking into account which and how many QoS policies can be associated with any one entity as defined by DDS.

There have been several decisions made regarding the design of the DQML metamodel that affects DQML's functionality and how it is used. These involve scope, constraints, and the relationship of QoS policies to DDS entities. To focus specifically on the QoS policy configuration challenges outlined above, the scope of DQML is limited to only those DDS entities that can have QoS policies associated with them. In addition to Data Reader, Data Writer, and Topic outlined before, 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 a factory for DDS entities for a particular domain or logical network, and (4) Domain Participant Factories, which are factories for generating Domain Participants. While other entities and constructs exist in DDS none of them directly use QoS policies and are therefore not included within the scope of DQML.

The constraints placed on QoS policies for compatibility and consistency are defined in the DDS specification. DQML uses the Object Constraint Language (OCL)[13] implementation provided by GME to define these constraints. As noted in Section 3.1, Challenge 1, 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 included in DQML. The constraints are checked when explicitly prompted by the user.

4.2 Functionality of DQML

DQML allows DDS application developers to specify and control the following aspects of QoS policy configuration:

Creation of DDS entities. DQML allows developers to create the DDS entities involved with QoS policy configuration. DQML supports the seven DDS entities that can be associated with QoS policies.

Creation of DDS QoS policies. DQML also allows developers to create the DDS QoS policies involved with QoS policy configuration. DQML supports the 22 DDS policies that can be associated with entities to provide the required QoS along with the attributes, the appropriate ranges of values, and defaults.

Creation of associations between DDS entities and QoS policies. DQML supports the generation of associations between the entities and the QoS policies and ensures that the associations are valid.

Checking compatibility and consistency constraints. DQML supports checking for compatible and consistent QoS policy configurations. The user initiates this checking and DQML reports if there are any violations.

4.3 Resolution of QoS Configuration Design Challenges

This section outlines how DQML is designed to address the challenges that arise when creating valid QoS policy configurations as outlined in Section 3.1.

4.3.1 Resolving QoS Compatibility

Challenge 1 in Section 3.1 describes the difficulties of ensuring compatible QoS policies between DDS entities that need to exchange data. DQML is designed to address this challenge by including compatibility checking in the modeling language itself. As shown in Figure 4, DQML users can invoke compatibility checking to make sure that the QoS policy configuration specified is valid. If QoS policies are found to be incompatible then the user is notified *at design time* and given details of the incompatibility.

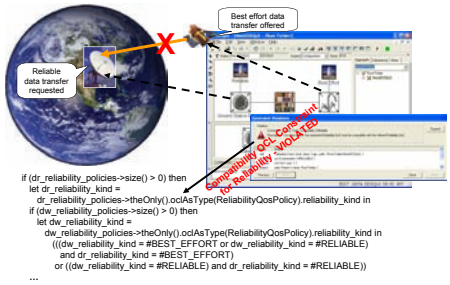

Figure 4: Example MMS QoS Policy Incompatibility

DQML is designed to allow for easy resolution of QoS incompatibilities. QoS policies are associated with DDS entities via connections made between them. When an incompatibility is found it can be quickly resolved in most cases by associating the incompatible DDS entities to the same QoS policy ensuring compatibility.

4.3.2 Resolving QoS Consistency

Challenge 2 in Section 3.1 describes the difficulties of ensuring consistent QoS policies so that the policies will not conflict with each other when associated with the same DDS entity. DQML is designed to address this challenge by including consistency checking in the modeling language itself. Just as with compatibility checking, the user of DQML can invoke consistency checking to ensure that the QoS policy configuration is valid. If inconsistent QoS policies are found then the user is notified *at design time* with detailed information to aid in correcting the problem.

Figure 5 shows an example of how DQML catches inconsistent QoS policies for a QoS policy configuration. In this example, the deadline period, *i.e.*, 10, is less than the time based filter's minimum separation, *i.e.*, 15. Both of these policies are associated with the same MMS Spacecraft data reader. DQML checks the consistency of the modeled QoS policies and notifies the user of the violation.

4.3.3 Transforming QoS Policy Configurations

Challenge 3 in Section 3.1 outlines the problems of propagating a valid QoS policy configuration from design to im-

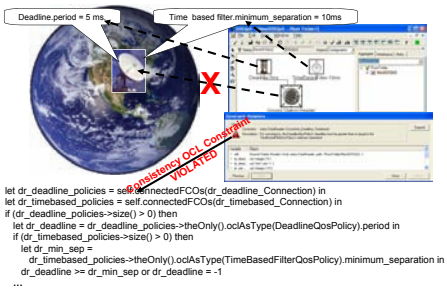

Figure 5: Example MMS QoS Policy Inconsistency

plementation and then deployment. DQML addresses this challenge by enabling its developers to create model interpreters that can iterate over the QoS policy configuration model designed in DQML to create appropriate implementation artifacts (*e.g.*, source code, configuration files) that will faithfully recreate the QoS policy configuration as designed.

```
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
```

Figure 6: Example QoS Policy Configuration File

Figure 6 shows an example of a QoS policy configuration file for an MMS Spacecraft data reader as generated by DQML. In this listing, QoS policies associated with the data reader along with values for the policies are shown. This file can then be directly plugged into the implementation of MMS to ensure the desired QoS configuration.

5. CONCLUDING REMARKS

Highly configurable pub/sub middleware platforms are increasingly being used as the basis for event-based distributed systems. This adoption has also increased demand for powerful QoS polices. We developed DQML to address the challenges of managing the complexity of the wide variety of possible QoS policies, attributes, and values available with DDS, though its DSML techniques can also be generalized to other pub/sub middleware, such as JMS.

The following is a summary of lessons learned from our experience using DQML to model QoS policy configurations for DDS that are guiding our future work.

• **Ensuring semantic compatibility of QoS policies is crucial to proper deployment of event-based pub/-sub systems.** When incompatibilities exist the system will not perform as designed with potentially cascading effects of non-performance and unpredictability.

• **Ensuring that the designed QoS policy configuration is faithfully mapped to deployment is crucial to system integrity.** Even with the valid and proper design of a QoS policy configuration, a system will not perform as intended if the configuration is not faithfully transformed into the implementation. In future work we plan to incorporate DQML into model-based deployment tools [3] to provide "correct-by-construction" deployment for DDS.

• **Becoming proficient with OCL is hard and it can be easy to misuse.** Most systems developers are not familiar with rule-based languages such as OCL so there is the added overhead of training. Development tool support for OCL is often rudimentary, *e.g.*, little debugging support, which lowers productivity. In future work we therefore plan to address enforcing constraints by looking at other constraint solving technologies, such as the Constraint Logic Programming Finite Domain (CLP(FD)) solver [6].

• **Real-world use and feedback are crucial elements to developing a robust DSML.** As with any development tool, DSMLs can only be as good as the particular environments and systems with which they are used. DSML designers cannot foresee all the potential environments and scenarios *a priori*. In future work, we therefore plan to use DQML to generate QoS policy configurations for benchmarking of DDS implementations and the various DDS QoS policies.

6. REFERENCES

[1] S. Curtis. The Magnetospheric Multiscale Mission...Resolving Fundamental Processes in Space Plasmas. *NASA STI/Recon Technical Report N*, pages 48257–+, December 1999.

[2] Akos Ledeczi et al. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[3] Gan Deng et al. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In *Proceedings of the 3rd Working Conference on Component Deployment*, Grenoble, France, November 2005.

[4] 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.

[5] 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.

[6] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[7] Object Management Group. *Event Service Specification Version 1.1*, OMG Document formal/01-03-01 edition, March 2001.

[8] Object Management Group. *Data Distribution Service for Real-time Systems Specification*, 1.2 edition, January 2007.

[9] Organization for the Advancement of Structured Information Standards. *Web Services Base Notification Version 1.3*, OASIS Document wsn-ws_base_notification-1.3-spec-os edition, October 2006.

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

[11] SUN. Java Messaging Service Specification. `java.sun.com/products/jms/`, 2002.

[12] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, April 1997.

[13] 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.