Quantitative Productivity Analysis of a Domain-Specific Modeling Language

Joe Hoffert
Indiana Wesleyan University, USA

Douglas C. Schmidt
Vanderbilt University, USA

Aniruddha Gokhale
Vanderbilt University, USA

ABSTRACT

Model-driven engineering (MDE), in general, and Domain-Specific Modeling Languages (DSMLs), in particular, are increasingly used to manage the complexity of developing applications in various domains. Although many DSML benefits are qualitative (*e.g.*, ease of use, familiarity of domain concepts), there is a need to quantitatively demonstrate the benefits of DSMLs (*e.g.*, quantify when DSMLs provide savings in development time) to simplify comparison and evaluation. This chapter describes how we conducted quantitative productivity analysis for a DSML (*i.e.*, the Distributed Quality-of-Service (QoS) Modeling Language (DQML)). Our analysis shows (1) the significant quantitative productivity gain achieved when using a DSML to develop configuration models compared with not using a DSML, (2) the significant quantitative productivity gain achieved when using a DSML interpreter to automatically generate implementation artifacts as compared to alternative methods when configuring application entities, and (3) the viability of quantitative productivity metrics for DSMLs.

INTRODUCTION

Model-driven engineering (MDE) helps address the problems of designing, implementing, and integrating applications (Hästbacka, 2011)(Lukman, 2010)(Schmidt, 2006)(Hailpern, 2006)(Atkinson, 2003)(Kent, 2002). MDE is increasingly used in domains involving modeling software components, developing embedded software systems, and configuring quality-of-service (QoS) policies. Key benefits of MDE include (1) raising the level of abstraction to alleviate accidental complexities of low-level and heterogeneous software platforms, (2) more effectively expressing designer intent for concepts in a domain, and (3) enforcing domain-specific development constraints. Many documented benefits of MDE are qualitative, *e.g.*, use of domain-specific entities and associations that are familiar to domain experts, and visual programming interfaces where developers can manipulate icons representing domain-specific entities to simplify development. There is a lack of documented quantitative benefits for domain-specific modeling languages (DSMLs), however, that show how developers are more productive using MDE tools and how development using DSMLs yields fewer bugs.

Conventional techniques for quantifying the benefits of MDE in general (*e.g.*, comparing user-perceived usefulness of measurements for development complexity (Abrahao and Poels, 2007, 2009)) and DSMLs in particular (*e.g.*, comparing elapsed development time for a domain expert with and without the use of the DSML (Loyall, Ye, Shapiro, Neema, Mahadevan, Abdelwahed, Koets, & Varner, 2004)) involve labor-intensive and time-consuming experiments. For example, control and experimental groups of developers may be tasked to complete a development activity during which metrics are collected (*e.g.*, number of defects, time required to complete various tasks). These metrics also often require the analysis of domain experts, who may be unavailable in many production systems.

Even though DSML developers are typically responsible for showing productivity gains, they often lack the resources to demonstrate the quantitative benefits of their tools. One way to address this issue is via productivity analysis, which is a lightweight approach to quantitatively evaluating DSMLs that measures how productive developers are, and quantitatively exploring factors that influence productivity (Boehm, 1987) (Premraj, Shepperd, Kitchenham, & Forselius, 2005). This chapter applies quantitative productivity measurement using a case study of the Distributed QoS Modeling Language (DQML), which is a DSML for designing valid QoS policy configurations and transforming the configurations into correct-by-construction implementations. Our productivity analysis of DQML shows significant productivity gains compared with common alternatives, such as manual development using third-generation programming languages. While this chapter leverages DQML as a case study, in general the productivity gains and analysis presented are representative of DSMLs' ability to reduce accidental complexity and increase reusability.

The remainder of this chapter includes the following objectives: highlighting related work; presenting an overview of DQML; outlining the DQML case study as a basis for DSMLs; describing productivity analysis for DSMLs leveraging the DQML case study; and presenting concluding remarks and lessons learned.

BACKGROUND

This section presents related work in the area of metrics for MDE and domain-specific technologies. We present work on quantitative analysis for MDE technologies as well as metrics to support quantitative evaluation.

Conway and Edwards (2004) focus on measuring quantifiable code size improvements using the NDL Device Language (NDL), which is a domain-specific language applicable to device drivers. NDL abstracts details of the device resources and constructs used to describe common device driver operations. The creators of NDL show quantitatively that NDL reduces code size of a semantically correct device driver by more than 50% with only a slight impact on performance. While quantifiable code size improvements are shown by using NDL, the type of improvement is applicable to DSLs where a higher level language is developed to bundle or encapsulate lower level, tedious, and error prone development. The productivity analysis for a DSL is easier to quantify since common units such as lines of source code are used. Conway and Edwards present compelling evidence of productivity gains of NDL although they do not encompass all the benefits of automatic code generation found with DSMLs such as the ease of a graphical user interface (GUI).

Bettin (2002) measures productivity for domain-specific modeling techniques within the domain of object-oriented user interfaces. Comparisons are made between (1) traditional software development where no explicit modeling is performed, (2) standard Unified Modeling Language (UML)-based software development, where UML is interpreted as a graphical notation providing a view into the source code, and (3) domain-specific notations to UML to support a higher-level abstraction that automatically generates large parts of the implementation. While the use of the domain-specific notations show a sharp reduction in the number of manually-written lines of source code as compared to traditional software development, the addition of modeling elements comes at some cost since no models are developed in traditional software development. The trade-off of the manual coding and modeling efforts is not clear quantitatively.

Balasubramanian, Schmidt, Molnar, & Ledeczi (2007) quantitatively analyze productivity gains within the context of the System Integration Modeling Language (SIML). SIML is a DSML that performs metamodel composition augmenting elements of existing DSMLs or adding additional elements. The productivity analysis of SIML focuses on the reduction of development steps needed for functional integration as compared to manual integration including design and implementation using native tools. The design and implementation steps are weighted more heavily (*i.e.*, are more expensive in development resources such as time and man-power) than using SIML which provides automated DSL integration. The analysis shows a 70% reduction in the number of distinct integration steps for a particular use case.

Genero, Piattini, Abrahao, Insfran, Carsi, & Ramos (2007) qualitatively measure ease of comprehension for class diagrams generated using various transformation rules via an experimental approach. From a given requirements model UML class diagrams were generated using 3 different sets of transformation rules. Human subjects were then asked to evaluate how easy the generated diagrams were to understand. While this experimental approach gleans valuable user feedback, this approach also incurs substantial experimental resources and time by involving human subjects and also targets the qualitative aspect of ease of understanding.

Abrahao and Poels (2007) have created OO-Method Function Points (OOmFP) which enhance function point analysis (FPA), originally designed for functional user requirements, to be applicable to object-oriented technologies. The metric generates a value related to the amount of business functionality a system provides to a user. Experimental procedures were conducted with students to compare FPA and OOmFP. The experiments showed that OOmFP consumed more time than FPA but the results were more accurate and more easily reproducible. Abrahoa and Poels (2009) then extended their OOmFP work into the area of Web applications which they termed OOmFPWeb. OOmFPWeb was designed to evaluate functionality of Web systems based on user-defined requirements encapsulated in the conceptual model of an application developed for the Web rather than based solely on implementation artifacts created once the application had been fully developed (Cleary, 2000)(Reifer, 2000).

Martínez, Cachero, & Meliá (2012) created and implemented empirical productivity studies when using a traditional code-centric approach, a UML model-based approach (describe what model-based means), and a model-driven engineering approach. The objective of the experiments was to compare the productivity of junior Web developers when developing the business layer of a Web 2.0 application. The experiments included 26 web application students that were grouped into six teams. The results of the experiments showed statistically significant increases in productivity when the subjects leveraged MDE practices. However, the results come with several caveats. While productivity improvements are empirically shown, the authors acknowledge that the results have not been extrapolated to a wider scope, *e.g.*, level of subject expertise, the domain of interest, number of developers. The results are only applicable to the scope stated in the research and quantifying the results for a different level of development expertise or for a different domain are non-trivial and challenging.

Papotti, do Prado, de Souza, Cirilo, & Pires (2013) developed an empirical experiment to compare a model-driven development approach where code is automatically generated to the classic life-cycle development approach where code is manually generated. The goal of the experiment was to conduct a comparative analysis of the amount of time spent in developing Create, Retrieve, Update, and Delete (CRUD) functionality for web systems using application entity classes described in UML class diagrams. In particular, the subjects were working on a medium to large size system to store academic information. In this experiment, the developers were 19 upper-level computer science and computer engineering students at the Federal University of São Carlos in Brazil. The subjects were divided into 9 homogeneous groups based on the level of development experience. The results showed that using automatic code generation decreases development time in a statistically significant way (*i.e.*, reduction of 90.98%). However, no results were presented as to other factors typically of interest (*e.g.*, quality of the code). The researchers also acknowledge that the results are limited to CRUD web applications using university students as software developers and that additional experiments would need to be conducted to gather relevant information for other developer populations (*e.g.*, larger or smaller groups, more or less experienced developers, different software development domains).

Mohagheghi, Gilani, Stefanescu, Fernandez, Nordmoen, & Fritzsche (2013) present experience reports from three large industrial regarding the benefits of model-driven engineering. The software applications involved range from SAP's large-scale enterprise business applications to Telefonica's network modeling to WesternGeco's simulation of seismic instrumentation to discover oil and gas below the sub-sea surface. The three participating organizations found benefits to using MDE techniques especially in the areas of providing abstractions of complex systems from different levels of abstraction and from different perspectives. The domain specific models also facilitated communication with non-technical experts and aided in simulation, testing, and performance analysis. The negative impacts of MDE were also noted such as the extra effort and negative impact on tools when developing reusable solutions. Moreover, merging several tools with one another required several transformations. The authors note that productivity improvements could not be measured although the potential exists based on feedback from the users. However, no approaches for quantitatively measuring the productivity improvements were proposed nor were classification of relevant factors for productivity improvements (*e.g.*, size of development team).

In contrast to the work outlined above, this chapter showcases (1) a quantitative productivity metric of developing DSMLs and (2) a quantitative productivity metric of DSML interpreters that transform models into implementation artifacts. These metrics are important since DSML developers and interpreter developers need to understand not only the quantitative benefit of DSMLs and interpreters but also the development effort for which the DSML and interpreter are justified.

QUANTITATIVE PRODUCTIVITY ANALYSIS

DSML Case Study: Overview of the Distributed QoS Modeling Language (DQML)

The *Distributed QoS Modeling Language* (DQML) is a DSML that addresses key inherent and accidental complexities of ensuring semantically compatible QoS policy configurations for publish/subscribe (pub/sub) middleware. Semantic compatibility 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. DQML automates the analysis and synthesis of semantically compatible QoS policy configurations for the OMG *Data Distribution Service* (DDS), which is an open standard for QoS-enabled pub/sub middleware (Object Management Group, 2007). DQML was developed using the *Generic Modeling Environment* (GME) (Ledeczi, Bakay, Maroti, Volgyesi, Nordstrom, Sprinkle, & Karsai, 2001), which is a metaprogrammable environment for developing DSMLs.

This section provides an overview of DDS and the structure and functionality of DQML. Although DQML focused initially on QoS policy configurations for DDS, the approach can be applied to other pub/sub technologies, such as Web Services Brokered Notification (OASIS, 2006), Java Message Service (Sun Microsystems, 2002), CORBA Event Service (Object Management Group, 2004-1), and CORBA Notification Services (Object Management Group, 2004-2).

A: Overview of the OMG Data Distribution Service (DDS)

DDS defines a standard pub/sub architecture and runtime capabilities that enables applications to exchange data in event-based distributed systems. DDS provides efficient, scalable, predictable, and resource-aware data distribution via its *Data-Centric Publish/Subscribe* (DCPS) layer, which supports a global data store where publishers write and subscribers read data, respectively. Its modular structure, power, and flexibility stem from its support for (1) *location-independence*, via anonymous pub/sub, (2) *redundancy*, by allowing any numbers of readers and writers, (3) *real-time QoS*, via its 22 QoS policies, (4) *platform-independence*, by supporting a platform-independent model for data definition that can be mapped to different platform-specific models, and (5) *interoperability*, by specifying a standardized protocol for exchanging data between distributed publishers and subscribers.

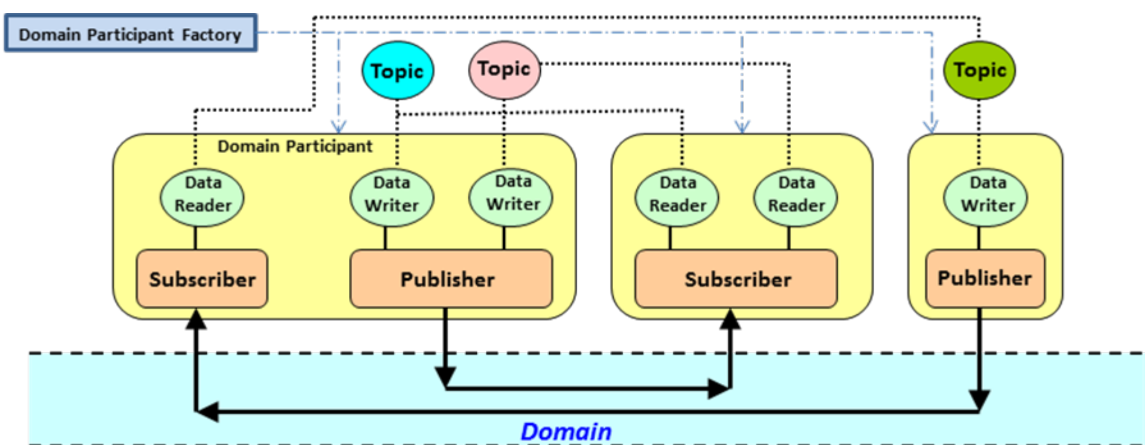

Figure 1: Architecture of the DDS Data-Centric Publish/Subscribe (DCPS) Layer

As shown in Figure 1, several types of DCPS entities are specified for DDS. A *domain* represents the set of applications that communicate with each other. A domain acts likes a virtual private network so that DDS entities in different domains are completely unaware of each other even if on the same machine or in the same process. A *domain participant factory*'s sole purpose is to create and destroy domain participants. The factory is a pre-existing singleton object that can be accessed by means of the *get_instance()* class operation on the factory. A *domain participant* provides (1) a container for all DDS entities for an application within a single domain, (2) a factory for creating publisher, subscriber, and topic entities, and (3) administration services in the domain, such as allowing the application to ignore locally any information about particular DDS entities.

DDS Entity	Description
Data Reader	Subscribes to the data of particular topics
Data Writer	Publishes the data for particular topics
Domain Participant	Creates Publisher, Subscriber, and Topic entities
Domain Participant Factory	Creates and destroys domain participant entities
Publisher	Manages one or more data writers
Subscriber	Manages one or more data readers
Topic	Describes the type and structure of the data to read or write

Table 1: DDS Entities Relevant to QoS Properties

DDS is topic-based, which allows strongly typed data dissemination since the type of the data is known throughout the entire system.[i] As outlined in Table 1, DDS *topic* describes the type and structure of the data to read or write, a *data reader* subscribes to the data of particular topics, and a *data writer* publishes data for particular topics. Various properties of these entities can be configured using combinations of the 22 QoS policies that are described in Table 2. In addition, *publishers* manage one or more data writers while *subscribers* manage one or more data readers. Publishers and subscribers can aggregate data from multiple data writers and readers for efficient transmission of data across a network.

DDS QoS Policy	Description
Deadline	Determines rate at which periodic data should be refreshed
Destination Order	Determines whether data writer or data reader determines order of received data
Durability	Determines if data outlives the time when written or read
Durability Service	Details how data that can outlive a writer, process, or session is stored
Entity Factory	Determines enabling of DDS entities when created
Group Data	Attaches application data to publishers, subscribers
History	Sets how much data is kept for data readers
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	Determines if multiple data writers can write to the same topic instance
Ownership Strength	Sets ownership of topic instance 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 dissemination
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

Topic types are defined via the OMG Interface Definition Language (IDL) that enables platform-independent type definition. An IDL topic type can be mapped to platform-specific native data types, such as C++ running on VxWorks or Java running on real-time Linux. Below we show an example topic definition in IDL that defines an analog sensor with a sensor id of type *string* and a value of type *float*.

```
struct AnalogSensor {
  string sensor_id; // key
  float value; // other sensor data
};
```

DDS provides a rich set of QoS policies, as illustrated in Table **2**. Each QoS policy has ~2 attributes, with most attributes having an unbounded number of potential values, *e.g.*, an attribute of type character string or integer. The DDS specification defines which QoS policies are applicable for certain entities, as well as which combinations of QoS policy values are semantically compatible. For example, if a data reader and data writer associated via a common topic want data to flow reliably, they must both specify reliable transmission via the reliability QoS policy.

The extensive QoS support of DDS and the flexibility of the QoS policies present the challenges of appropriately managing the policies to form the desired QoS configuration. These challenges not only include ensuring valid QoS parameter types and values but also ensuring valid interactions between the policies and the DDS entities. Moreover, managing semantic compatibility increases the accidental complexity of creating valid QoS configuration since not all valid combinations of QoS policies will produce the desired system behavior as outlined above with the flow of reliable data.

DSMLs can help address these challenges. DSMLs can reduce the variability complexity of managing multiple QoS policies and their parameters by presenting the QoS policies as modeling elements that are automatically checked for appropriate associations and whose parameters are automatically typed and checked for appropriate values. DSMLs can also codify constraints for semantic compatibility to ensure that data flows as intended. Moreover, DSMLs can automatically generate implementation artifacts that accurately reflect the design.

B. Structure of the DQML Metamodel

DDS defines 22 QoS policies shown in Table 2 that control the behavior of DDS applications. DQML models all of these DDS QoS policies, as well as the seven DDS entities (*i.e.*, *Data Reader*, *Data Writer*, *Topic*, *Publisher*, *Subscriber*, *Domain Participant*, and *Domain Participant Factory)* that can have QoS policies. 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. 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) (Warmer & Kleppe, 2003) (Cabot & Gogolla, 2012) implementation provided by GME to define these constraints. Compatibility constraints involve a single type of QoS policy (*e.g.*, reliability QoS policy) associated with more than one type of DDS entity (*e.g.*, data reader, data writer) whereas consistency constraints involve a single DDS entity (*e.g.*, data reader) with more than one QoS policy (*e.g.*, deadline and latency budget QoS policies). Both types of constraints are included in DQML.

Figure 2 shows an example of how OCL is used in DQML to determine if deadline properties between data readers and data writers are compatible. Line 1 determines if a deadline policy has been specified for a data reader. If so, then the period for that deadline is retrieved and stored in the variable dr_deadline (*i.e.*, period for data reader's deadline) on line 2. Line 4 determines if a deadline policy has been specified for a data writer. If so, then the period for that deadline is retrieved and stored in the variable dw_deadline (i.e., period for the data writer's deadline) on line 5. Lines 7 through 11 are comments meant to add clarification to the constraint code. Line 12 compares the period of the deadline for the data writer to the period of the deadline for the data reader and returns that values OR'ed with whether or not the data reader deadline was infinite as specified on line 13. We used the value of -1 as a convention to specify an infinite deadline period (which is the default value for a deadline period). If the data reader's deadline period is infinite then it is compatible with whatever deadline period the data writer specifies (per the DDS specification). Line 16 checks to see if the data reader's deadline period was infinite which will match the default data writer's deadline period of infinite since in this case no deadline property was associated with the data writer. Finally, line 14 returns true since no deadline property was specified for either the data writer or the data reader. In this case the deadlines are compatible since the default period values are infinite for both the data reader and data writer.

```
1  if (dr_deadline_policies->size() = 1) then
2     let dr_deadline = dr_deadline_policies->theOnly().oclAsType(DeadlineQosPolicy).period in
3
4     if (dw_deadline_policies->size() = 1) then
5        let dw_deadline = dw_deadline_policies->theOnly().oclAsType(DeadlineQosPolicy).period in
6
7     -- The DDS specification defines compatibility such that the data writer's offered deadline
8     -- must be less than or equal to the data reader's requested deadline.
9     -- We have chosen -1 to represent infinity.
10    -- NOTE: A Topic can use this function as well since there is an implied DataWriter if
11    --       a Topic is associated with a DurabilityQosPolicy
12       dw_deadline <= dr_deadline or
13       dr_deadline = -1
14    else
15       -- Here compare dr_deadline <= infinity
16       dr_deadline = -1
17    endif
18 else
19    -- Here compare infinity <= infinity
20    true
21 endif
```

Figure 2: OCL Example for Checking Deadline Compatibility

There are important differences between a programming language (*e.g.*, C++, Java, and Python) and a constraint language. The two types of languages address different needs (*i.e.*, specifying constraints vs. specifying program execution). However, while OCL is not a programming language OCL does provide programmers with common programming language features (*e.g.*, if/then/else code blocks) as illustrated in Figure 2. Moreover, it can be helpful for programmers to think of OCL as a language to define predicates (*i.e.*, functions to return either true or false) since predicates are a common programming language construct particularly with respect to generic/template programming (Sutton and Stroustrup, 2011). Therefore, codifying constraints using a constraint language like OCL is equivalent to codifying program execution in terms of the effort required and the complexity involved.

The OCL constraints for a DQML model are checked when explicitly initiated by the user. Programmatically checking these constraints greatly reduces the accidental complexity of creating valid QoS configurations. The DSML design decision to have constraint checking explicitly initiated by the user rather than automatically initiated by the DSML was made so that model developers could develop and save partial or incomplete models as development checkpoints.

C. Functionality of DQML

DQML allows DDS application developers to specify and control key aspects of QoS policy configuration in the following ways.

Creation of DDS entities. As illustrated in Figure 3, 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.

Figure 3: DDS Entities Supported in DQML

Creation of DDS QoS policies. DQML allows developers to create the 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. As shown in Figure 4, DQML ameliorates the variability complexity of specifying (1) valid associations between QoS policies and DDS entities and (2) valid QoS policy parameters, parameter types, and values (including default values).

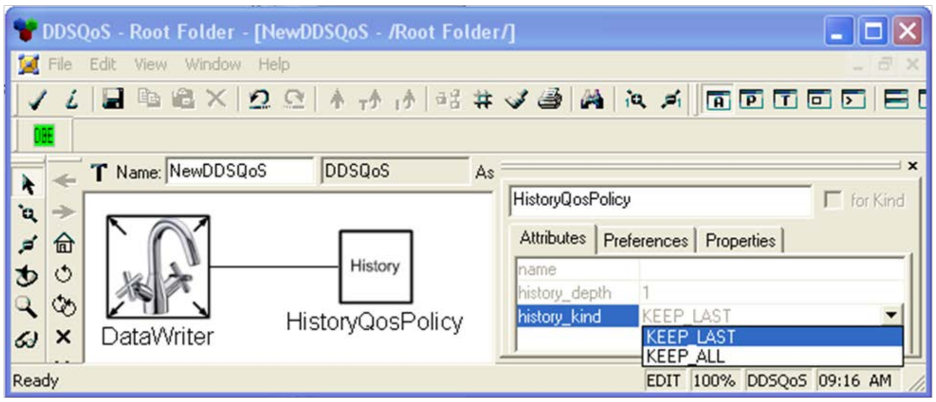

Figure 4: Example of DQML QoS Policy Variability Management

Creation of associations between DDS entities and QoS policies. As shown in Figure 5, DQML supports the generation of associations between the entities and the QoS policies and ensures that the associations are valid. DQML's support of correct associations is important since only certain types of entities can be associated with certain other entities and only certain types of QoS policies can be associated with certain types of entities.

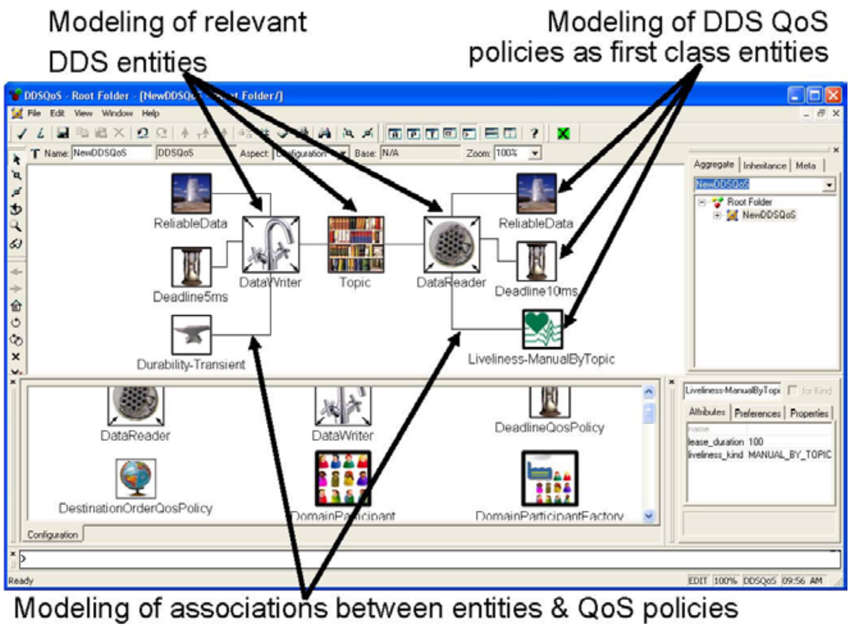

Figure 5: Modeling Entities, Policies, and Association in DQML

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. Figure 6 shows DQML detecting and notifying users of incompatible reliability QoS policies while Figure 7 shows inconsistency between a deadline's period and a time based filter's minimum separation both associated with the same data reader.

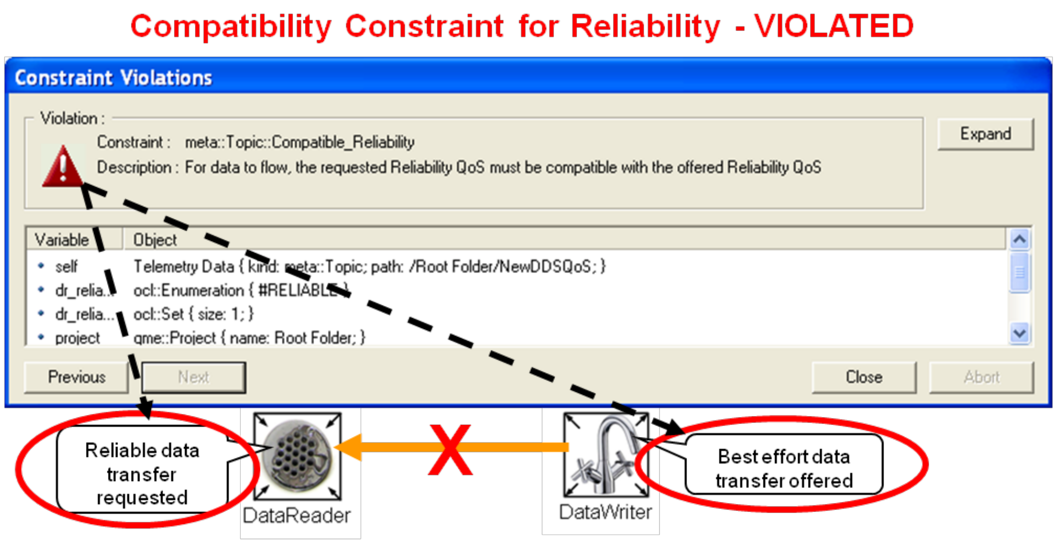

Figure 6: Example of DQML QoS Policy Compatibility Constraint Checking

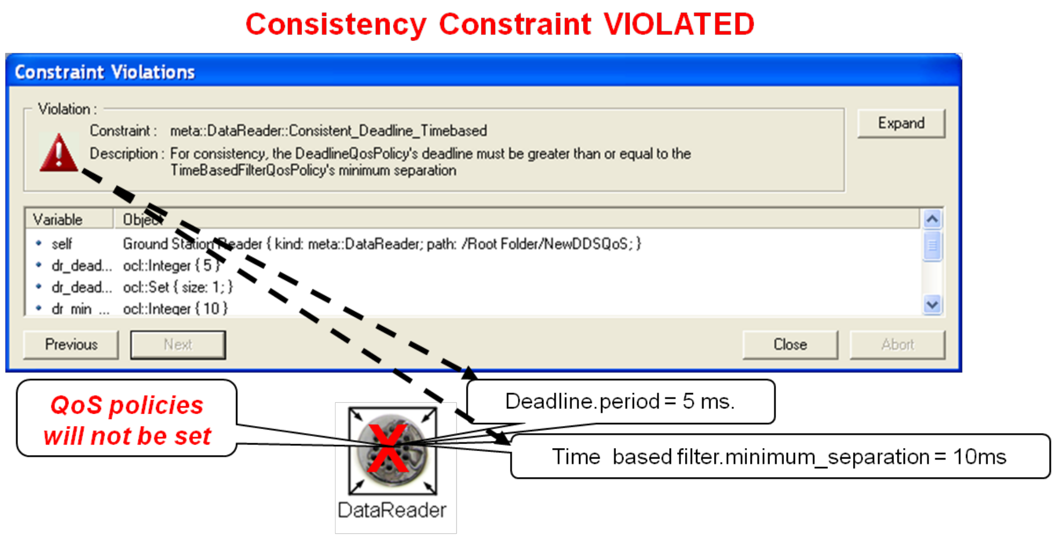

Figure 7: Example of DQML QoS Policy Consistency Constraint Checking

<u>Transforming QoS policy configurations from design to implementation.</u> DQML transforms QoS policy configurations into implementation artifacts via application specific interpreters. Figure 8 shows a representative implementation artifact for a data reader and two data writers. At runtime the DDS middleware will then read this XML while deploying and configuring the DDS entities.

```
- <DQML>
  - <DataWriter name="DataWriter1">
      <deadline>period="50"</deadline>
    </DataWriter>
  - <DataReader name="DataReader1">
      <deadline>period="100"</deadline>
      <timebased_filter>min_separation="0"</timebased_filter>
    </DataReader>
  - <DataWriter name="DataWriter2">
      <reliability>kind="RELIABLE" max_blocking_time="100"</reliability>
    </DataWriter>
  </DQML>
```

Figure 8: Example QoS Policy Configuration File

DQML Case Study: DDS Benchmarking Environment (DBE)

Developing DDS applications is hard due to inherent and accidental complexities. The inherent complexities stem from determining appropriate configurations for the DDS entities. The accidental complexities stem from managing the variability, semantic compatibility, and transformation of QoS configurations. This section presents a case study highlighting development complexity to show how DQML can be applied to improve productivity compared to manual approaches.

At least eight different implementations of DDS are available each with its own set of strengths and market discriminators.[ii] A systematic benchmarking environment is needed to objectively evaluate the QoS of these implementations. Such evaluations can also help guide the addition of new features to the DDS standard as it evolves. The *DDS Benchmarking Environment* (DBE) (www.dre.vanderbilt.edu/DDS/html/dbe.html) tool suite was developed to examine and evaluate the QoS of DDS implementations (Xiong, Parsons, Edmondson, Nguyen, & Schmidt, 2007). DBE is an open-source framework for automating and managing the complexity of evaluating DDS implementations with various QoS configurations. DBE consists of a repository containing scripts, configuration files, test ids, test results, a hierarchy of Perl scripts to automate evaluation setup and execution, and a shared C++ library for collecting results and generating statistics.

We use DBE as a case study in this chapter to highlight the challenges of developing correct and valid QoS configurations, as well as to analyze the productivity benefits of DQML. Although we focus on DBE in our case study, production DDS-based applications will generally encounter the same accidental complexities when implementing QoS parameter settings, *e.g.*, design-to-implementation transformation fidelity; valid, correct, compatible, and consistent settings. DDS QoS policy settings are typically specified for a DDS implementation programmatically by manually creating source code in a third-generation computer language, *e.g.*, Java or C++. Manual creation can incur the same accidental complexities as the DBE case study without the integration of MDE tools like DQML.

Since DDS has a large QoS configuration space (as outlined in the DDS overview above) there is an exponential number of testing configurations where QoS parameters can vary in several orthogonal dimensions. Manually performing evaluations for each QoS configuration, DDS implementation, and platform incurs significant accidental complexity. Moreover, the effort to manage and organize test results also grows dramatically along with the number of distinct QoS configurations.

```
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 9: Example Portion of a DBE QoS Settings File

DBE deploys a QoS policy configuration file for each data reader and data writer. As shown in Figure 9, the files contain simple text with a line-for-line mapping of QoS parameters to values, *e.g.*, *datawriter.deadline.period=10*. A file is associated with a particular data reader or data writer. For DBE to function properly, QoS policy settings in the configuration files must be correct to ensure that data flows as expected. If the QoS policy configuration is invalid, incompatible, inconsistent, or not implemented as designed, the QoS evaluations will not execute properly.

The DBE configuration files have traditionally been hand-generated using a text editor, which is tedious and error-prone since the aggregate parameter settings must ensure the fidelity of the QoS configuration design as well as the validity, correctness, compatibility, and consistency with respect to other values. Moreover, the configuration files must be managed appropriately, *e.g.*, via unique and descriptive filenames, to ensure the implemented QoS parameter settings reflect the desired QoS parameter settings. To address these issues, we developed an interpreter for DBE within DQML to automate the production of DBE QoS settings files.

When applying DQML to generate a QoS configuration for DBE we model (1) the desired DDS entities, (2) the desired QoS policies, (3) the associations among entities, and (4) the associations between entities and QoS policies. After an initial configuration is modeled, we then perform constraint checking to ensure compatible and consistent configurations. Other constraint checking is automatically enforced by the DQML metamodel as a model is constructed (*e.g.*, listing only the parameters applicable to a selected QoS when modifying values, allowing only valid values for parameter types).

We then invoke the DBE interpreter to generate the appropriate QoS settings files. These files contain the correct-by-construction parameter settings automatically generated by the interpreter as it traverses the model and transforms the QoS policies from design to implementation. Finally, we execute DBE to deploy data readers and data writers using the generated QoS settings files and run experiments to collect performance metrics.

DSML Productivity analysis

This section provides a lightweight taxonomy of approaches to developing quantitative productivity analysis for a DSML. It also presents a productivity analysis for DQML that evaluates implementing QoS configurations for the DBE case study from the previous section.

A. Productivity analysis Approach

When analyzing productivity gains for a given DSML, analysts can employ several different types of strategies, such as

- **Design development effort,** comparing the effort (*e.g.*, time, number of design steps (Balasubramanian Schmidt, Molnar, & Ledeczi, 2007)) or number of modeling elements (Kavimandan & Gokhale, 2008) (von Pilgrim, 2007)) it takes a developer to generate a design using traditional methods (*e.g.*, manually) versus generating a design using the DSML,

- **Implementation development effort**, comparing the effort (*e.g.*, time, lines of code) it takes a developer to generate implementation artifacts using traditional methods, *i.e.*, manual generation, versus generating implementation artifacts using the DSML,

- **Design quality**, comparing the number of defects in a model or an application developed traditionally to the number of defects in a model or application developed using the DSML (Kärnä, Tolvanen, & Kelly, 2009),

- **Required developer experience**, comparing the amount of experience a developer needs to develop a model or application using traditional methods to the amount of experience needed when using a DSML, and

- **Solution exploration**, comparing the number of viable solutions considered for a particular problem in a set period of time using the DSML as compared to traditional methods or other DSMLs (White, Schmidt, Nechypurenko, & Wuchner, 2008).

Our focus is on the general area of quantitative productivity measurement (*e.g.*, number of entities and relationships to manage in a configuration, implementation development effort in terms of lines of code). The remainder of this section compares (1) number of DDS entities and relationships between those entities when considering development of a DSML and (2) the lines of configuration code manually generated for DBE data readers and data writers to the lines of C++ code needed to implement the DQML DBE interpreter, which in turn generates the lines of configuration code automatically.

B. Metrics and Productivity Analysis for Developing a DSML

In this section we analyze the effect on productivity and the breakeven point of developing a DSML. For this analysis we use the case study of DQML as a representative DSML. A DSML is used to design domain-specific elements such as configuring QoS for applications that are incorporating DDS to support QoS. However, the effort to develop a DSML is not without some cost and effort. We compare the effort to develop a DSML to designing the domain-specific elements without the use of a DSML. This analysis is helpful in determining whether or not to develop a DSML. Using the context of DQML as a case study this section specifically focuses on quantifying the effort to design QoS configurations with and without the use of a DSML. Using the DSML productivity analysis taxonomy outlined in the previous section, this section focuses on the category of *design development effort*.

To design domain-specific configurations the designer must be familiar with all the relevant elements in the domain. For the DDS case study of DQML, the seven relevant QoS entity elements are listed in Table 1. Knowledge of these seven DDS elements is needed to understand possible DDS QoS designs whether a DSML is being developed or not. Moreover, the designer needs to have familiarity with all seven entity elements even if not all the elements are used for a particular design. This familiarity ensures that all the relevant DDS entity elements are included in design considerations and that no relevant DDS entity elements are excluded.

The effort to incorporate knowledge of these DDS elements into the development of a DSML includes knowledge of the meta-modeling environment being used. In this case study GME was used to develop the DQML metamodel. However, the analysis performed for developing a DSML applies equally to other DSML development environments such as the Eclipse Modeling Framework (EMF) (Gronback, 2009) (Budinsky, Steinberg, Merks, 2009) since the general tasks of managing the types of entities and managing the valid associations between them must be performed regardless of any tool used. GME is a metaprogrammable development environment that is used to develop DSMLs. GME provides a GUI with drag-and-drop capabilities for creating relevant metamodel entities. The cost to learn a metamodel development tool is non-trivial. However, with the exception of simple point solutions as outlined below, a QoS designer will want to maintain designs in some electronic form (*e.g.*, utilizing a drawing package) since non-electronic forms are harder to manage (*e.g.*, copy, index, store). This management step will ensure that (1) design enhancements and modifications can be saved and archived and (2) design intent can be clearly communicated to the development team including the original designer.

A single developer creating QoS configurations for DDS might be able to manage all the relevant information without the use of external tools (*i.e.*, in his or her head). However, the threshold for the point at which a developer can manage all this information is typically quite low. Research has shown that the number of items a person can mentally manage is between 4 and 9 items (Miller, 1956)(Cowan, 2001)(Kamiński, Brzezicka, & Wróbel, 2011). For our analysis we use the more conservative value of 9 items. Therefore, if the single developer must manage more than 9 different items including, for example, any relevant DDS entities and DDS QoS policies then use of a DSML is warranted.

This threshold of 9 items applies to more than simply the unique types of entities (*e.g.*, 22 unique QoS polices in DDS) since it also applies to the total number of entities even if some entities are of the same type (*e.g.*, more than 9 instances of the Data Reader entity). The role that each entity plays in the overall design along with the relationships between these entities add to the inherent design complexity. Moreover, this threshold of 9 items also applies to the relationships that the entities can have. For example, as shown in Figure 6, a DDS reliability QoS policy is only applicable between a data reader entity and a data writer entity (or alternatively between a topic entity and a data reader entity or between a topic entity and a data writer entity). In this case, the QoS configuration developer must not only manage the three DDS entities but also the valid relationships between them.

If the developer is working in a team then the number of communication paths between the team members must also be managed to ensure correct information. As shown by Tsui, Karam, & Bernal (2013), the number of communication paths between members of a development team increases geometrically as the number of team members increases linearly. If the number of communication paths is more than 9 then a development tool with clearly defined semantics is warranted in order to clearly communicate the design being developed. For a group of 5 developers all working together the number of communication paths surpasses the threshold of 9 (*i.e.*, 10 communication paths).

If a simple electronic format is utilized (*e.g.*, a simple text file), the cost of developing a format to describe a design or the cost of unambiguously describing the design in text will surpass the effort to learn a metamodeling tool such as GME or EMF. This proposition will be true for all cases that involve any combination of more than 9 (1) types of entities, (2) modeled entities even of a single type, (3) relationships between the modeled entities, or (3) number of developers involved. Moreover, if the implementation artifacts are going to be maintained or modified then developing a DSML is warranted. Short-term memory ranges from minutes to hours while long-term memory ranges from days to years (Bailey, Bartsch, & Kandel, 1996). We use the conservative value of one week to denote the time by which short-term memory has been lost. Using this value of one week, a DSML is warranted if a design is going to be maintained beyond the timeframe of one week. Figure 10 outlines the decision process to determine if developing a DSML is warranted.

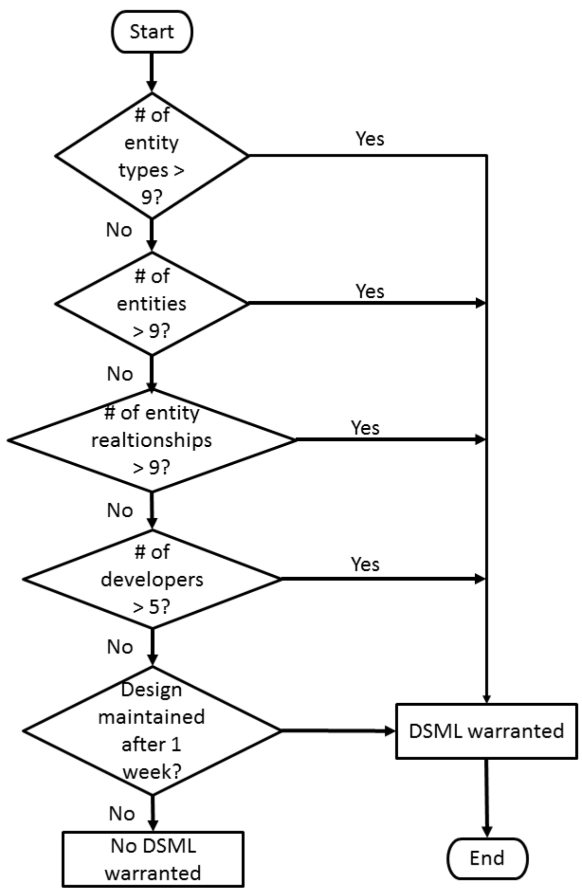

Figure 10: DSML Decision Process

In the case of DDS, the specification is fairly straightforward as to how entities relate to each other and what QoS properties are applicable to which entities. The specification also provides UML class diagrams to document these relationships. Therefore, a DSML for DDS QoS configurations is quite amenable to development using a metamodeling tool such as GME or EMF. Other domains and specifications for those domains might not be as compatible with class diagrams. However, in general entities and relationships between those entities will be needed and which map well to classes and class diagrams.

Additionally, DDS provides 22 QoS property elements (as outlined in Table 2) with which the QoS designer must be familiar. As outlined previously regarding knowledge of all DDS entity elements, a QoS designer will need to have at least cursory knowledge of these 22 QoS property elements so that the relevant QoS property elements are included in the design and that no relevant QoS property elements are excluded. This knowledge is needed regardless if a DSML is being developed.

Recent research by Pati, Feiock, and Hill (2012) outlines how to incorporate proactive modeling so that the complexities of developing valid models can be ameliorated. This approach can reduce the overhead of developing models and make the creation of a DSML more appealing. This approach has not yet been analyzed for its quantitative benefit in model generation but is a promising area of research to explore in this regard.

C. Metrics and Productivity Analysis for Developing a DSML Interpreter

Once a DSML has been developed, analysis should be performed to determine if development of an interpreter for the DSML is warranted. Below we analyze the effect on productivity and the breakeven point of using a DSML interpreter for QoS policy configurations. In particular, this section quantifiably analyzes the productivity of developing and using a DSML interpreter to generate implementation artifacts for DBE compared to manual generation. Although configurations can be *designed* using various methods as outlined in previous work (Hoffert, Schmidt, & Gokhale, 2007), manual *implementation* of configurations is applicable to these other design solutions since these solutions provide no guidance for implementation. Using the DSML productivity analysis taxonomy outlined in the previous section, this section focuses on the category of *implementation development effort*.

Within the context of DQML, we developed an interpreter specific to DBE to support DBE's requirement of correct QoS policy configurations. The interpreter generates QoS policy parameter settings files for the data readers and data writers that DBE configures and deploys. All relevant QoS policy parameter settings from a DQML model are output for the data readers and data writers including settings from default as well as explicitly assigned parameters.

As appropriate for DBE, the interpreter generates a single QoS policy parameter settings file for every data reader or data writer modeled. Care is taken 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. Moreover, the interpreter's generation of filenames aids in QoS settings files management (as described previously) since the files are uniquely and descriptively named. The following subsections detail the scope, development effort, and productivity analysis of DQML's DBE interpreter versus manual methods.

1. Scope. DBE uses DDS data readers and data writers. Our productivity analysis therefore focuses on these entities and, in particular, the QoS parameters relevant to them. In general, the same type of analysis can be performed for other DDS entities for which QoS policies can be associated. As shown in Table 3, 15 QoS policies with a total of 25 parameters can be associated with a single data writer.

QoS Policy	**Number of Parameters**	**Parameter 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	boolean
Total Parameters	**25**	

Table 3: DDS QoS Policies for Data Writers

QoS Policy	**Number of Parameters**	**Parameter 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	2	2 ints
Reliability	2	1 enum, 1 int
Resource Limits	3	3 ints
Time Based Filter	1	int
User Data	1	string
Total Parameters	**18**	

Table 4: DDS QoS Policies for Data Readers

Likewise, Table 4 shows 12 QoS policies with a total of 18 parameters that can be associated with a single data reader. Within the context of DBE, therefore, the total number of relevant QoS parameters is **18 + 25 = 43**. Each QoS policy parameter setting (including the parameter and its value) for a data reader or writer corresponds to a single line in the QoS parameter settings file for DBE.

2. Interpreter development. We developed the DBE interpreter for DQML using GME's Builder Object Network (BON2) framework, which provides C++ code to traverse the DQML model utilizing the Visitor pattern. When using BON2, developers of a DSML interpreter only need to modify and add a small subset of the framework code to traverse and appropriately process the particular DSML model. More specifically, the BON2 framework supplies a C++ visitor class with virtual methods (*e.g.*, visitModelImpl, visitConnectionImpl, visitAtomImpl). The interpreter developer then subclasses and overrides the applicable virtual methods.

The DDS entities relevant to DQML are referred to as *model implementations* in BON2. Therefore, the DBE interpreter only needs to override the *visitModelImpl()* method and is not concerned with other available virtual methods. When the BON2 framework invokes *visitModelImpl()* it passes a model implementation as an argument. A model implementation includes methods to (1) traverse the associations a DDS entity has (using the *getConnEnds()* method) and specify the relevant QoS policy association as an input parameter (*e.g.*, the association between a data writer and a deadline QoS Policy), (2) retrieve the associated 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 utilizes 160 C++ statements within the BON2 framework. We stress that any interpreter development is a one-time cost; specifically there is no development cost for the DBE interpreter since it is already developed. The main challenge in using BON2 is understanding how to traverse the model and access the desired information. After interpreter developers are familiar with BON2, the interpreter development is fairly straightforward. We detail the steps of developing the DBE interpreter below.

```
1  class DDSQoSVisitor : public Visitor {
2  public:
3      DDSQoSVisitor ();
4      ~DDSQoSVisitor ();
5
6  protected :
7      virtual void visitAtomImpl (const Atom& atom);
8      virtual void visitModelImpl (const Model& model);
9      virtual void visitConnectionImpl ( const Connection& connection);
10
11     void processDataReaderQos (const Model& dataReader);
12     void processDataWriterQos (const Model& dataWriter);
13
14     void outputDDSEntityQos (const Model& dds_entity,
15                              const std::string &entity_name,
16                              const std::string &entity_abbrev,
17                              const std::string &qos_connection_name,
18                              const std::string &qos_name,
19                              const std::map<std::string, std::string> &attribute_map,
20                              int entity_count,
21                              bool &file_opened,
22                              std::ofstream &out_file);
23
24 private:
25     std::ofstream out_file_;
26 };
```

Figure 11: Visitor Class for DBE Interpreter

Figure 11 outlines the visitor class that has been created for the DBE interpreter for use within the BON2 framework. This class is the only class that needs to be implemented for the DBE interpreter. Line 1 determines the class name and its derivation from the BON2 Visitor class. Lines 3 and 4 declare the default constructor and destructor respectively. Lines 7 – 9 declare the abstract methods visitAtomImpl, visitModelImpl, and visitConnectionImpl inherited from the BON2 Visitor class that need to be defined for the DBE interpreter. Lines 11 and 12 declare methods to process data readers and data writers respectively. Lines 14 – 22 declare the main method that processes the QoS properties for a data reader or data writer and writes the QoS parameters to the appropriate file. Line 25 defines the debugging output file that had been used for debugging the DBE interpreter.

As is shown in Figure 11, the structure of the DBE visitor class is fairly simple and straightforward. Moreover, of the three methods inherited from the BON2 Visitor class and declared on lines 7 – 9 only the visitModelImpl method declared on line 8 is a non-empty method. For DBE, the only DQML entities of interest are what GME terms the *model elements* which for DBE's interests are the data readers and data writers. The DBE interpreter is not concerned with traversing atom or connection elements since these elements will be addressed by processing the model elements.

We now focus on the implementations of the relevant methods particularly as they relate to complexity and required background knowledge. The default constructor and destructor simply open and close the file used for debugging which is not required functionality for the DBE interpreter. Therefore the implementations of these two methods (which total two C++ statements) are excluded to save space. The visitAtomImpl and visitConnectionImpl methods are defined (since the inherited methods are abstract) but empty (since they are not needed).

As shown in Figure 12, the visitModelImpl method determines the type of model element currently being processed and calls the appropriate method, *i.e.*, processDataReaderQos for a data reader on line 6 and processDataWriterQos for a data writer on line 12. The lines written to out_file_ are simply for debugging purposes and are not required by DBE. The DBE interpreter developer required familiarity with the DQML metamodel to know the names of the model elements of interest but the model elements in the metamodel were given intuitive names to reduce accidental complexity, *e.g.*, DataReader and DataWriter on lines 3 and 9 respectively.

```
1   void DDSQoSVisitor::visitModelImpl( const Model& model )
2   {
3       if (model->getModelMeta().name() == "DataReader")
4       {
5           out_file_ << "DDS DataReader Name: " << model->getName() << std::endl;
6           processDataReaderQos(model);
7           out_file_ << "...Done DDS DataReader Name: " << model->getName() << std::endl;
8       }
9       else if (model->getModelMeta().name() == "DataWriter")
10      {
11          out_file_ << "DDS DataWriter Name: " << model->getName() << std::endl;
12          processDataWriterQos(model);
13          out_file_ << "...Done DDS DataWriter Name: " << model->getName() << std::endl;
14      }
15  }
```

Figure 12: visitModelImpl Method

```
1  void DDSQoSVisitor::processDataWriterQos( const Model& dataWriter )
2  {
3      static int dw_count = 1;
4      const std::string dw_name("DataWriter");
5      const std::string dw_prefix("DW");
6      std::ofstream output_file;
7      bool file_opened = false;
8      std::map<std::string, std::string> attrib_map;
9
10     // Handle Deadline QoS Policy
11     attrib_map.clear ();
12     attrib_map["period"] = "datawriter.deadline.period=";
13     outputDDSEntityQos (dataWriter,
14                         dw_name,
15                         dw_prefix,
16                         "dw_deadline_Connection",
17                         "Deadline",
18                         attrib_map,
19                         dw_count,
20                         file_opened,
21                         output_file);
22
23     // Handle History QoS Policy
24     attrib_map.clear ();
25     attrib_map["history_kind"] = "datawriter.history.kind=";
26     attrib_map["history_depth"] = "datawriter.history.depth=";
27     outputDDSEntityQos (dataWriter,
28                         dw_name,
29                         dw_prefix,
30                         "dw_history_Connection",
31                         "History",
32                         attrib_map,
33                         dw_count,
34                         file_opened,
35                         output_file);
       •
       •
       •
```

Figure 13: processDataWriterQos Method

Figure 13 outlines the processDataWriterQos method. For each QoS policy applicable to a data writer this method sets up a mapping of DQML QoS parameter names to DBE QoS parameters names. Then the method calls outputDDSEntityQoS method to write the QoS parameter values to the appropriate file. The interpreter developer needed to have an understanding of the QoS parameter names for DBE, the QoS parameter names in the DQML metamodel, and the names of the associations between data readers/writers and QoS policies in the DQML metamodel. However, as with the model elements in the DQML metamodel, the QoS parameters were given intuitive names to reduce accidental complexity, *e.g.*, history_kind and history_depth on lines 25 and 26 respectively, as were the connection names, *e.g.*, dw_deadline_Connection and dw_history_Connection on lines 16 and 30 respectively.

Figure 13 shows the source code for processing the deadline and history QoS policies. The rest of the method which has been elided for brevity handles all the other QoS policies relevant to data writers. Finally, the method closes the QoS parameter file if one has been opened previously and increments the count of data writers processed so that unique filenames can be generated. Likewise, the processDataReaderQos method provides the same functionality for QoS policies and parameters relevant to data readers. Its source code is not included due to space constraints.

Figure 14 presents the outputDDSEntityQos method which traverses the connection that a data reader or data writer has to a particular QoS policy (*e.g.*, connections to QoS policies for data readers or data writers) and writes the QoS parameters out to the QoS settings file for that data reader or writer. Lines 14 – 21 and 54 – 57 provide error checking for the BON2 framework and have been elided for space considerations. Line 11 retrieves the associations that the data reader or writer has with a particular QoS policy, *e.g.*, all the associations between a data reader and the reliability QoS policy. Lines 24-27 retrieve the endpoint of the connection which will be the associated QoS policy of the type specified as the input parameter of line 4. Lines 29 and 30 retrieve the parameters of the associated QoS policy, lines 31 – 41 open a uniquely named DBE QoS settings file if one is not currently open, and lines 42 – 52 iterate through the QoS parameters and write them out to the opened file in the required DBE format using the attribute mapping passed as an input parameter on line 6.

```
1   void DDSQoSVisitor::outputDDSEntityQos (const Model& dds_entity,
2                                           const std::string &entity_name, // e.g., "DataReader"
3                                           const std::string &entity_abbrev, // e.g., "DR"
4                                           const std::string &qos_connection_name,
5                                           const std::string &qos_name,
6                                           const std::map<std::string, std::string> &attribute_map,
7                                           int entity_count,
8                                           bool &file_opened,
9                                           std::ofstream &out_file)
10  {
11      std::multiset<ConnectionEnd> conns = dds_entity->getConnEnds(qos_connection_name);
12      if (conns.size() > 0)
13      {
14          if (conns.size() > 1){ ... }
22          else
23          {
24              std::multiset<ConnectionEnd>::const_iterator iter(conns.begin ());
25              ConnectionEnd endPt = *iter;
26              FCO fco(endPt);
27              if (fco)
28              {
29                  std::set<Attribute> attrs = fco->getAttributes ();
30                  std::set<Attribute>::const_iterator attr_iter(attrs.begin());
31                  if (!file_opened)
32                  {
33                      file_opened = true;
34                      std::string filename;
35
36                      char cnt_buf [10];
37                      ::sprintf_s (cnt_buf, "%d", entity_count);
38                      std::string cnt_str = cnt_buf;
39                      filename = entity_abbrev + cnt_str + "_" + dds_entity->getName () + ".txt";
40                      out_file.open(filename.c_str ());
41                  }
42                  for (; attr_iter != attrs.end (); ++attr_iter)
43                  {
44                      Attribute attr = *attr_iter;
45                      std::string attr_name = attr->getAttributeMeta ().name ();
46                      std::map<std::string, std::string>::const_iterator map_iter =
47                          attribute_map.find (attr_name);
48                      if (map_iter != attribute_map.end ())
49                      {
50                          out_file << map_iter->second << attr->getStringValue () << std::endl;
51                      }
52                  }
53              }
54              else{ ... }
58          }
59      }
60  }
```

Figure 14: outputDDSEntityQos Method

Since the BON2 framework relies on the Visitor pattern, familiarity with this pattern can be helpful. This familiarity is not required, however, and developers minimally only need to implement relevant methods for the automatically generated Visitor subclass. In general, the DQML interpreter code specific to DBE (1) traverses the model to gather applicable information, (2) creates the QoS settings files, and (3) outputs the settings into the QoS settings files.

The C++ development effort for DQML's DBE interpreter is only needed one time. In particular, no QoS policy configuration developed via DQML for DBE incurs this development overhead since the interpreter has already been developed. The development effort metrics of 160 C++ statements are included *only* to be used in comparing manually implemented QoS policy configurations.

3. Comparing manually developing DBE implementation artifacts. To compare model-driven engineering approaches in general and the DQML DBE interpreter in particular, we outline the steps to generate the implementation artifacts of DBE QoS settings files given a manually generated QoS configuration design. Several areas of inherent and accidental complexity need to be addressed for manual development of DBE QoS settings files. To illustrate these complexities we follow the steps needed to transform a data reader entity associated with a reliability QoS policy from design into implementation. We assume the QoS configuration design is specified either in a text or graphics file or handwritten. We also assume that the QoS configuration design has been developed separately from the generation of implementation artifacts to separate these concerns and divide the labor.

Variability Complexity. Implementation developers must ensure the correct semantics for the association between the data reader and the reliability QoS policy. Developers cannot assume that the data reader, the reliability QoS policy, or the association between the two are valid and correctly specified since the configuration was manually generated. The data reader and reliability QoS policy must be cross-referenced with the DDS specification. This cross-referencing entails checking that (1) a data reader can be associated with a reliability QoS policy, (2) the parameter names specified for the reliability QoS policy are appropriate (*e.g.*, only kind and max_blocking_time are valid reliability QoS parameters), and (3) the values for the parameters are valid (*e.g.*, only RELIABLE and BEST_EFFORT are valid values for the reliability kind). Moreover, developers must manage the complexity of creating a separate QoS settings file for the data reader and ensuring a unique and descriptive filename that DBE can use.

Semantic Compatibility Complexity. Implementation developers must ensure the correct consistency semantics for the data reader's reliability QoS policy and the other QoS policies associated with the data reader. If QoS policies associated with the data reader are inconsistent then the policies cannot be used. Moreover, the developer must ensure correct semantics for the data reader's reliability QoS policy and data writers associated with the same topic. If QoS policies associated with the data reader are incompatible then the data will not be received by the data reader.

For the reliability QoS policy there are no inconsistency concerns. Developers must verify that this is the case, however, by checking the DDS specification for consistency rules. For the reliability QoS policy there are potential incompatibilities. If the reliability QoS policy kind for the data reader is specified as RELIABLE the developer must traverse the QoS configuration and check the reliability QoS policies for *all* data writers associated with the same topic. If no associated data writer has a reliability QoS kind set to RELIABLE (either explicitly or implicitly via default values) then the data reader can never receive any data thereby making the data reader superfluous. Default values for QoS parameters must therefore be known and evaluated for compatibility even if not explicitly specified. Manually traversing the QoS configuration to check for compatibility and accounting for default parameter values is tedious and error prone and greatly exacerbates the accidental complexity of generating implementation artifacts.

Faithful Transformation. Implementation developers must ensure that the QoS configuration design is accurately mapped to the implementation artifacts appropriate for DBE. As noted above, this transformation includes creating and managing a QoS settings file for each data reader and writer. Moreover, the developer must ensure that the syntax of QoS settings conform to what DBE requires. For example, the reliability's maximum blocking time of 10 ms must be specified as datareader.reliability.max_blocking_time=10 on a single line by itself in the QoS settings file for the particular data reader.

4. Analysis. The hardest aspect of developing DQML's DBE interpreter is traversing the model's data reader and data writer elements along with the associated QoS policy elements using the BON2 framework. Conversely, the most challenging aspects of manually implementing QoS policy configurations are (1) maintaining a global view of the model to ensure compatibility and consistency, (2) verifying the number, type, and valid values for the parameters of the applicable QoS policies, and (3) faithfully transforming the configuration design into implementation artifacts. On average, implementing a single C++ statement for the DBE interpreter is no harder than implementing a single parameter statement for the DBE QoS settings files. When implementing a non-trivial QoS policy configuration, therefore, development of the C++ code for the DBE interpreter is no more challenging than manually ensuring that the QoS settings in settings files are valid, consistent, compatible, and correctly represent the designed configuration. Below we provide additional detail into what can be considered a non-trivial QoS policy configuration.

The development and use of the DBE interpreter for DQML is justified for a *single* QoS policy configuration when at least 160 QoS policy parameter settings are involved. These parameter settings correlate to the 160 C++ statements for DQML's DBE interpreter. Using the results for QoS parameters in Table 3 and Table 4 for data readers and data writers, Figure 15 shows the justification for interpreter development. The development is justified with ~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 = 165 QoS policy parameter settings). For comparison, the breakeven point for data reader/writer pairs is **3.72** (*i.e.*, 160/43).

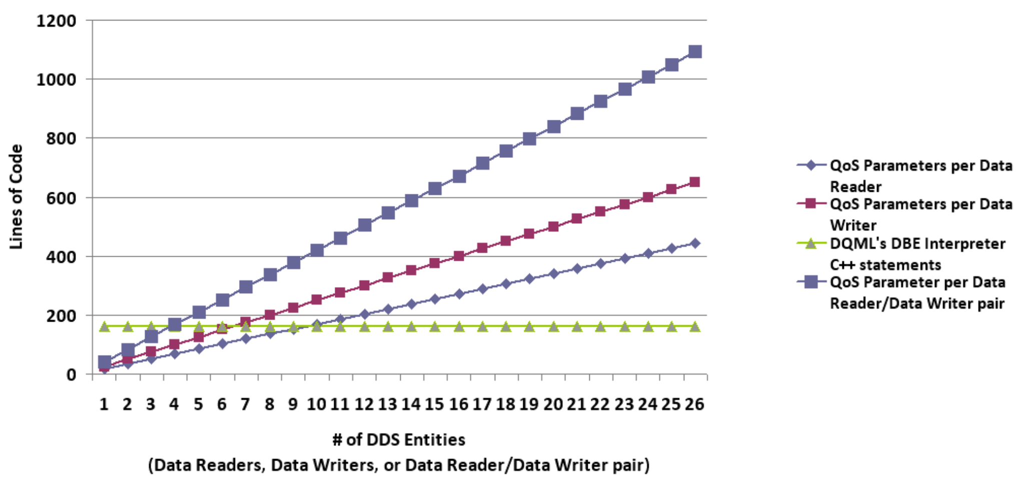

Figure 15: Metrics for Manual Configuration vs. DQML's Interpreter

We also quantified the development effort needed to support topics if the DBE interpreter required that functionality. Table 5 shows the DDS QoS policies and policy parameters applicable to topics. To support topics an additional 59 C++ statements would need to be added. Conversely, for manual generation 23 more QoS parameters need to be considered for each topic. The breakeven point for data reader/writer/topic triplets becomes **3.32** (*i.e.*, (160 + 59)/(43 + 23)) which is less than the breakeven point for data reader/writers alone (*i.e.*, 3.72).

This breakeven point is less because the additional source code to support topics can leverage existing code, in particular, the outputDDSEntityQos method outlined in Figure 14. The breakeven point can be applicable for *any* interpreter that leverages the commonality of formatting regardless of the entity type (cf. outputDDSEntityQos method). Moreover, the complexity of developing any DQML interpreter is lessened by having the DBE interpreter as a guide. The design and code of the DBE interpreter can be reused by another application-specific interpreter to navigate a DQML model and access the QoS policies.

QoS Policy	**Number of Parameters**	**Parameter 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
Reliability	2	1 enum, 1 int
Resource Limits	3	3 ints
Transport Priority	1	int
Topic Data	1	string
Total Parameters	**23**	

Table 5: DDS QoS Policies for Topics

Table 6 also shows productivity gains as a percentage for various numbers of data readers and data writers. The percentage gains are calculated by dividing the number of parameter values for the data readers and data writers involved by the number of interpreter C++ statements, *i.e.*, 160, and subtracting 1 to account for the baseline manual implementation (*i.e.*, ((# of data reader and writer parameters)/160)-1). The gains increase faster than the increase in the number of data readers and data writers (*e.g.*, the gain for 10 data readers and data writers is more than twice as much for 5 data readers and data writers) showing that productivity gains are greater when more entities are involved.

# of Data Readers and Data Writers (each)	**Total # of Parameters**	**Productivity Gain**
5	215	34%
10	430	169%
20	860	438%
40	1720	975%
80	3440	2050%

Table 6: Productivity Gains using DQML’s DBE Interpreter

The interpreter justification analysis shown relates to implementing a single QoS policy configuration. The analysis includes neither the scenario of modifying an existing valid configuration nor the scenario of implementing new configurations for DBE where no modifications to the interpreter code would be required. Changes made even to an existing valid configuration 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, the parameters of the various QoS policies being modified. These challenges are as applicable when changing an already valid QoS policy configuration as they are when creating an initial configuration. Moreover, the complexity for developing a new interpreter for some other application is ameliorated by having the DBE interpreter as a template for traversing a model in BON2.

In large-scale DDS systems (*e.g.*, shipboard computing, air-traffic management, and scientific space missions) there may be thousands of data readers and writers. As a point of reference with 1,000 data readers and 1,000 data writers, the number of QoS parameters to manage is *43,000* (*i.e.*, 18 * 1,000 + 25 * 1,000). 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.*, 43,000 / 160 = **268.75**).

The productivity analysis approach taken for DQML's DBE interpreter is applicable to other DSMLs since the complexities involved will be similar. A break-even point for the development effort of an interpreter for any DSML will exist. We outline four areas that directly influence this break-even point: number of entities, complexity of the entities, complexity of associations between the entities, and level of maintainability needed.

The number of entities affects the break-even point for interpreter development since the more entities that are to be considered the less likely any one individual will be able to manage these entities appropriately as outlined in the previous section. The guideline of 4 to 9 items that a human can process at one time can be helpful in exploring the break-even point for interpreter development. If there are more than 9 entities to be considered then the accidental complexity increases since the developer must manage the entities using some tool or device (*e.g.*, a piece of paper, a database) external to the person. With this external management comes the possibility of introducing errors in the use of the management tool (*e.g.*, incorrectly transcribing the entities from the developer's head to the tool).

Likewise, this same analysis holds for the complexity of entities as determined by the number of fields or parameters. If an entity contains more than 9 fields then some external tool should be used to manage this complexity. The use of a tool introduces accidental complexity (*e.g.*, incorrectly transcribing the order, names, or types of the parameters). The same analysis can also be applied to the number of associations made between entities to determine that complexity as well as the number of times a configuration will need to be modified.

If any one of these four areas exceeds the threshold of 9 then an interpreter is warranted. If more than one of these areas exceeds the threshold (*e.g.*, more than 9 entities with more than 9 associations between the entities) then the break-even point for an interpreter is lowered. The exact determination for justifying interpreter development will vary according to the application but the guidelines presented can provide coarse-grained justification.

FUTURE RESEARCH DIRECTIONS

Our future work for DQML includes assembly and deployment support as an aid for complex QoS configurations. Additionally, future research work includes developing domain-specific QoS profiles for different types of applications within the domain. These profiles would ease development of QoS configurations since they would provide a template or starting point for an application-specific configuration

DQML is available as open-source software and is included as part of the Component Synthesis with Model Integrated Computing (CoSMIC) tool chain being developed and maintained at Vanderbilt

University's Institute for Software Integrated Systems. Information regarding downloading, building, and installing CoSMIC can be found at www.dre.vanderbilt.edu/cosmic.

CONCLUSION

Although MDE and DSMLs have become increasingly popular, concrete evidence is needed to support the quantitative evaluation of DSMLs. This chapter described various approaches to quantitatively evaluating DSMLs via productivity analysis. We applied one of these approaches to a case study involving the *Distributed QoS Modeling Language* (DQML). The following is a summary of the lessons learned from our experience developing DQML and conducting productivity analysis using it for the DBE case study:

- **Trade-offs and the break-even point for DSMLs must be clearly understood and communicated.** There are pros and cons to any technical approach including DSMLs. The use of DSMLs may not be appropriate for every case and these cases must be evaluated to provide balanced and objective analysis. For a DSML product line the advantages of DSMLs will typically outweigh the development costs. For a one-time point solution the development of a DSML may not be justified, depending on the complexity of the domain.

- **The context for DSML productivity analysis should be well defined.** Broad generalizations of a DSML being "X" times better than some other technology is not particularly helpful for comparison and evaluation. A representative case study can be useful to provide a concrete context for productivity analysis.

- **Provide analysis for as minimal or conservative a scenario as possible.** Using a minimal scenario in productivity analysis allows developers to extrapolate to larger scenarios where the DSML use will be justified. This work highlighted the exponential benefit of automatically generating implementation artifacts from domain-specific models as the number of modeling elements increases linearly.

- **In general, the threshold for developing a DSML and accompanying interpreters is fairly low.** Several factors account for this low threshold including (1) the number of entities to be modeled, (2) the number of relationships between the modeled entities, (3) the number of developers involved, and (4) the likelihood of maintaining or modifying any one model. However, for situations where none of the above factors are relevant (e.g., the number of entities, relationships between entities, and number of developers is less than 7) then the overhead of learning a tool to develop a DSML and then developing the DSML might not be warranted.

KEY TERMS AND DEFINITIONS

Domain specific modeling language (DSML): A computer language (typically graphical in nature) developed for a particular domain (*e.g.*, disease diagnosis, QoS configuration).

DDS Benchmarking Environment: An open-source tool suite developed to examine and evaluate the QoS of various DDS implementations.

Data Distribution Service (DDS): An open standard for QoS-enabled pub/sub middleware supported by the Object Management Group (OMG).

Middleware: A software abstraction layer that typically resides between the operating system and the application layer (*e.g.*, the Java Virtual Machine).

Quality of service (QoS): properties of a software system related to non-functional aspects (*e.g.*, latency, reliability, usability).

Publish/Subscribe (pub/sub): An architectural paradigm for software systems that is data-centric (i.e., data is sent out and received) rather than call-centric (*i.e.*, request/response typically done by invoking a function and receiving a response).

Productivity Analysis: Analyzing the use of various software system development techniques to show productivity gains or losses (*i.e.*, typically concerned with the productivity of the software developers).

REFERENCES

Abrahao, S., & Poels, G. (2009). A family of experiments to evaluate a functional size measurement procedure for web applications. *Journal of Systems and Software*, *82*(2), 253-269.

Abrahao, S., & Poels, G. (2007). Experimental evaluation of an object-oriented function point measurement procedure. *Information and Software Technology*, *49*(4), 366-380.

Atkinson, C., & Kuhne, T. (2003). Model-driven development: A metamodeling foundation. *IEEE Software*, *20*(5), 36–41.

Bailey, C., Bartsch, D., & Kandel, E. (1996). Toward a molecular definition of long-term memory storage. *Proceedings of the National Academy of Sciences of the United States of America: Vol. 93* (Issue 24), (pp. 13445-13452). Stanford: Stanford University's HighWire Press.

Boehm, B. (1987). Improving Software Productivity. *Computer*, *20*(9), 43–57.

Bettin, J. (2002). Measuring the potential of domain-specific modeling techniques. 17*th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Seattle, WA.

Balasubramanian, K., Schmidt, D., Molnar, Z., & Ledeczi, A. (2007). Component-based system integration via (meta)model composition. *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems* (pp. 93–102). Los Alamitos: IEEE.

Bergmann, G., Hegedus, A., Horvath, A., Rath, I., Ujhelyi, Z., & Varro, D. (2011). Implementing efficient model validation in EMF tools. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (pp. 580–583). New York: ACM.

Budinsky, F., Steinberg, D., Merks, E. (2009). *EMF: Eclipse Modeling Framework*. Boston: Pearson Education, Inc.

Cabot, J., & Gogolla, M. (2012). Object Constraint Language (OCL): A Definitive Guide. *Proceedings of the 12th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-Driven Engineering* (pp.58–90). Berlin: Springer-Verlag.

Carzaniga, A., Rutherford, M., & Wolf, A. (2004). A routing scheme for content-based networking. *Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies, Vol. 2*, (pp. 918-928). Los Alamitos: IEEE.

Cleary, D., (2000). Web-based development and functional size measurement. *Proceedings of International Function Point Users Group Annual Conference*, San Diego, CA.

Conway, C. & Edwards, S. (2004). Ndl: a domain-specific language for device drivers. *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on languages, compilers, and tools for embedded systems* (pp. 30–36). New York: ACM.

Cowan, N. (2001). The magical number 4 in short-term memory: a reconsideration of mental storage capacity. *The Behavioral and Brain Sciences, 24*(1), 87-114.

Genero, M., Piattini, M., Abrahao, S., Insfran, E., Carsi, J., & Ramos, I. (2007). A controlled experiment for selecting transformations based on quality attributes in the context of MDA. *First International Symposium on Empirical Software Engineering and Measurement* (pp. 498-498). Los Alamitos: IEEE.

Gronback, R. (2009). *Eclipse Modeling Project: A Domain-Specific Language (Dsl) Toolkit*. Boston: Pearson Education, Inc.

Hailpern, B., & Tarr, P. (2006). Model-driven development: the good, the bad, and the ugly. *IBM Systems Journal*, *45*(3), 451-461.

Hästbacka, D., Vepsäläinen, T., and Kuikka, S. (2011). Model-driven development of industrial process control applications. *Journal of Systems and Software, 84*(7), 1100-1113.

Hoffert, J., Schmidt, D., & Gokhale, A. (2007). A QoS policy configuration modeling language for publish/subscribe middleware platforms. *Proceedings of the International Conference on Distributed Event-Based Systems* (pp. 140–145). New York: ACM.

Kamiński, J., Brzezicka, A., Wróbel, A (2011). Short-term memory capacity (7±2) predicted by theta to gamma cycle length ratio. *Neurobiology of Learning and Memory*, *95*(1), 19-23.

Kärnä, J., Tolvanen, J., & Kelly, S. (2009). Evaluating the use of domain-specific modeling in practice. *9th Workshop on Domain-Specific Modeling (OOPSLA 2009)*. New York: ACM.

Kavimandan, A., & Gokhale, A. (2008). Automated middleware QoS configuration techniques using model transformations. *Proceedings of the* 14*th IEEE Real-Time and Embedded Technology and Applications Symposium* (pp. 93–102). Los Alamitos: IEEE.

Kent, S. (2002). Model driven engineering. *Proceedings of the 3rd International Conference on Integrated Formal Methods* (pp. 286–298). Berlin: Springer.

Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., & Karsai, G. (2001). Composing domain-specific design environments. *Computer*, *34*(11), 44–51.

Li, G., & Jacobsen, H. (2005). Composite subscriptions in content-based publish/subscribe systems. *Proceedings of the 6th International Middleware Conference* (pp. 249-269). New York: Springer-Verlag.

Loyall, J., Ye, J., Shapiro, R., Neema, S., Mahadevan, N., Abdelwahed, S., Koets, M., & Varner, D. (2004). A Case Study in Applying QoS Adaptation and Model-Based Design to the Design-Time Optimization of Signal Analyzer Applications. *Military Communications Conference (MILCOM)* (pp. 1700–1705). Los Alamitos: IEEE.

Lukman, T., Godena, G., Gray, J., & Strmcnik, S. (2010). Model-driven engineering of industrial process control applications. *Proceedings of the 2010 IEEE Conference on Emerging Technologies and Factory Automation* (pp. 13-16). Los Alamitos: IEEE.

Martínez, Y., Cachero, C., & Meliá, S. (2012). Evaluating the impact of a model-driven web engineering approach on the productivity and the satisfaction of software development teams. *Lecture Notes in Computer Science*, Vol. 7387 (pp. 223-237). Berlin: Springer-Verlag.

Miller, G., (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, *63* (2), 81-97.

Mohagheghi, P., Gilani, W., Stefanescu, A., Fernandez, M., Nordmoen, B., and Fritzsche, M. (2013). Where does model-driven engineering help? Experiences from three industrial cases. *Software and Systems Modeling, 12*(3), 619-639.

OASIS. (2006). *Web services brokered notification 1.3*. Retrieved December 11, 2009 from http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-os.pdf.

Object Management Group. (2007). *Data distribution service for real-time systems, version 1.2*. Retrieved June 8, 2009, from http://www.omg.org/spec/DDS/1.2.

Object Management Group. (2004-1). *Event service specification version 1.2*. Retrieved December 11, 2009, from http://www.omg.org/cgi-bin/doc?formal/2004-10-02.

Object Management Group. (2004-2). *Notification service specification version 1.1*. Retrieved December 11, 2009, from http://www.omg.org/cgi-bin/doc?formal/2004-10-11.

Papotti, P., do Prado, A., de Souza, W., Cirilo, C., & Pires, L. (2013). A quantitative analysis of model-driven code generation through software experimentation. *Lecture Notes in Computer Science*, Vol. 7908 (pp. 321-337). Berlin: Springer-Verlag.

Pati, T., Feiock, D., & Hill, J. (2012). Proactive modeling: auto-generating models from their semantics and constraints. *Proceedings of the 2012 Workshop on Domain-specific modeling*, (pp. 7-12). New York: ACM.

Premraj, R., Shepperd, M., Kitchenham, B., & Forselius, P. (2005). An empirical analysis of software productivity over time, *Proceedings of the 11th IEEE International Symposium on Software Metrics* (pp. 37-46). Los Alamitos: IEEE.

Reifer, D., (2000). Web development: estimating quick-to-market software. *IEEE Software*, *17*(6), 57-64.

Schmidt, D., (2006). Model-driven engineering. *IEEE Computer*, *39*(2), 25-31.

Sun Microsystems. (2002). *Java Message Service version 1.1*. Retrieved December 11, 2009 from http://java.sun.com/products/jms/docs.html.

Sutton, A., & Stroustrup, B. (2011). Design of concept libraries for c++. *4th International Conference on Software Language Engineering* (pp. 97-118). Los Alamitos: IEEE.

Tsui, F., Karam, O., & Bernal, B. (2013). *Essentials of software engineering.* Burlington: Jones and Bartlett Publishers, Inc.

Pilgrim, J. (2008). Measuring the level of abstraction and detail of models in the context of mdd. *Models in Software Engineering: Workshops and Symposia at MoDELS 2007* (pp. 105–114). Berlin: Springer-Verlag.

Warmer, J., & Kleppe, A. (2003). *The object constraint language: getting your models ready for MDA.* Boston: Addison-Wesley Longman Publishing Co., Inc.

White, J., Schmidt, D., Nechypurenko, A., & Wuchner, E. (2008). Model intelligence: an approach to modeling guidance. *UPGRADE, 9*(2), 22–28.

Wu, Y., Hernandez, F., Ortega, F., Clarke, P., & France, R. (2010). Measuring the effort for creating and using domain-specific models. *10th Workshop on Domain-Specific Modeling*. (pp. 14:1-14:6). ACM, New York.

Xiong, M., Parsons, J., Edmondson, J., Nguyen, H., & Schmidt, D (2007). Evaluating technologies for tactical information management in net- centric systems. *Proceedings of the Defense Transformation and Net-Centric Systems conference*, Orlando, Florida.

[i] In contrast, content-based pub/sub middleware, such as Siena (Carzaniga, Rutherford, & Wolf, 2004) and the Publish/subscribe Applied to Distributed REsource Scheduling (PADRES) (Li & Jacobsen, 2005), examine events throughout the system to determine data types.
[ii] The Object Management Group maintains a list of vendors supplying DDS implementations at http://portals.omg.org/dds/category/web-links/vendors.