Improving Software Development Productivity for QoS Policy Configurations

Joe Hoffert and Douglas C. Schmidt
Vanderbilt University
Nashville, TN, USA, {jhoffert, schmidt}@dre.vanderbilt.edu

Introduction

Model-driven engineering (MDE) helps address the problems of designing, implementing, and integrating software applications. 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.*, the use of (1) domain-specific entities and associations that are familiar to domain experts and (2) visual programming interfaces where developers can manipulate icons representing domain-specific entities to simplify development. In general, however, there is a lack of documented quantitative benefits for *Domain-Specific Modeling Languages* (DSMLs) that show how (1) developers are more productive using MDE tools and (2) development using DSMLs yields fewer bugs. Conventional techniques for quantifying the benefits of DSMLs, such as comparing elapsed development time for a domain expert with and without the use of the DSML [1], 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 may also require the analysis of domain experts who are unavailable or otherwise engaged in production systems.

Although DSML developers are typically responsible for showing productivity gains, they often lack the resources to demonstrate the quantitative benefits of their tools. To address this issue, this chapter presents a lightweight approach to quantitatively evaluating DSMLs via *productivity analysis*, which measures how productive developers are and quantitatively explores factors that influence productivity [2][3]. This chapter focuses on applying quantitative productivity measurement on 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.

While there has been much prior work on domain-specific technologies, less attention has been focused on quantitative productivity metrics for DSMLs. Conway and Edwards [4] quantify code size improvements, but do not address key benefits of automatic code generation. Bettin [5] presents productivity analysis for domain-specific modeling techniques, although the trade-off of manual coding and modeling efforts is primarily qualitative. Balasubramanian *et al.* [6] provide quantitative productivity analysis of a DSML showing a reduction in the number of development steps for a particular use case, but do not address productivity gains over the life of the DSML. Our productivity analysis of DQML in this chapter shows it can provide significant productivity gains compared with common alternatives, such as manual development using third-generation programming languages.

Suggested Chapter Structure

The remainder of the chapter will be organized as follows.

Section 2 describes the *Distributed QoS Modeling Language* (DQML) which is a DSML that addresses key inherent and accidental complexities of ensuring semantically compatible QoS policy configurations for publish/subscribe (pub/sub) middleware. DQML initially focused on QoS policy configurations for the *Data Distribution Service* (DDS) [7], which is QoS-enabled pub-sub middleware standardized by the Object Management Group (OMG). DDS defines an anonymous pub/sub architecture to exchange data in

event-based distributed systems. The *data-centric pub/sub* (DCPS) layer of DDS provides a global data store where publishers write and subscribers read data. Its modular structure, power, and flexibility stem from its support for (1) *location-independence*, via anonymous publish/subscribe, (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.

DQML has been developed using the Generic Modeling Environment [8] (GME), which is a metaprogrammable environment for developing DSMLs. This section provides an overview of DQML's structure and functionality, focusing on the following topics:

- *Structure of the DQML Metamodel.* The DQML metamodel constrains the possible set of QoS policy configuration models that can be generated. The metamodel includes all 22 QoS policy types defined by DDS, as well as the seven DDS entity types that can have QoS policies associated with them.
- *Functionality of DQML.* DQML allows users to incorporate an arbitrary number of DDS entity instances from the seven entity types supported (*e.g.*, any number of data readers), as shown in Figure 1. DQML also allows users to specify DDS QoS policy instances (*e.g.*, deadline QoS policies). All DDS QoS policy parameters are supported along with the appropriate ranges of parameter values, as well as the default values. Users can modify parameter values as needed. DQML performs type checking on any modified parameters and will prohibit any invalid values (*e.g.*, assigning a character to an integer value). Moreover, for enumeration parameter types DQML presents only the appropriate enumeration values and allows the assignment of only one valid value to the parameter.

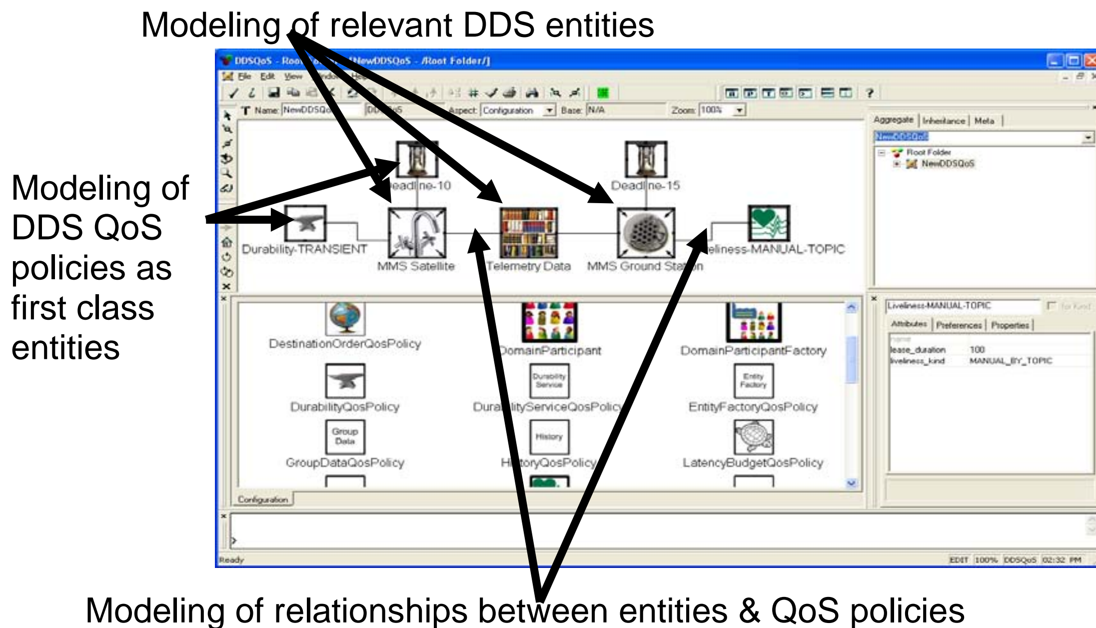

Fig. 1: The Distributed QoS Modeling Language (DQML)

Section 3 describes the DDS Benchmarking Environment (DBE) [9], which is a case study for DQML productivity analysis to highlight the challenges of developing correct and valid QoS configurations, as well as to analyze the productivity benefits of DQML. 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.

Although our case study focuses on DBE, production DDS-based applications will 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 and C++. Manual creation can thus incur the same accidental complexities as the DBE case study without the integration of MDE tools, such as DQML.

Section 4 taxonomizes 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 Section 3. Productivity gains for a given DSML can be analyzed in terms of several criteria, such as:

- *Design development effort***,** comparing the effort (*e.g.*, time, number of design steps [6], number of modeling elements [10], [11]) 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,
- *Required developer experience*, comparing the amount of experience a developer needs to generate 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.

The DQML productivity analysis focuses on the general area of quantitative productivity measurement—specifically on implementation development effort in terms of lines of code. The remainder of this section compares 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. We analyze the effect on productivity and the breakeven point of using DQML (as opposed to manual implementations of QoS policy configurations for DBE).

.Our productivity analysis focuses on DBE's use of DDS data reader and data writer 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 1, 15 QoS policies with a total of 25 parameters can be associated with a single data writer. Likewise, Table 2 shows 12 QoS policies with a total of 18 parameters 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 policy parameter settings file.

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 1: **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 2: **DDS QoS Policies for data readers**

The development and use of 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 Tables 1 and 2 for data readers and data writers, the development effort for the interpreter is justified with approximately 10 data readers, approximately 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). Further productivity gains are shown as the number of data readers and/or data writers increase.

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.

Section 5 presents concluding remarks and lessons learned, such as:

- *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.
- *The context for DSML productivity analysis needs to 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.

DQML is available as open-source software and can be downloaded in GME's XML format along with supporting files from www.dre.vanderbilt.edu/~jhoffert/DQML/DQML.zip.

References

1. J. Loyall *et al.*, "A Case Study in Applying QoS Adaptation and Model-Based Design to the Design-Time Optimization of Signal Analyzer Applications," in *Military Communications Conference (MILCOM)*, Monterey, California, Nov. 2004.
2. B. Boehm, "Improving Software Productivity," *Computer*, vol. 20, no. 9, pp. 43–57, Sept. 1987.
3. R. Premraj et al., "An Empirical Analysis of Software Productivity Over Time," *Software Metrics, 2005. 11th IEEE International Symposium*, Sept. 2005.
4. C. L. Conway and S. A. Edwards, "Ndl: A Domain-specific Language for Device Drivers," in *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. New York, NY, USA: ACM, 2004, pp. 30–36.
5. J. Bettin, "Measuring the Potential of Domain-specific Modeling Techniques," in *OOPSLA 2002: 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, WA, USA, November 2002.
6. K. Balasubramanian et al., "Component-based System Integration via (Meta)Model Composition," in *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 93–102.
7. Object Management Group. *Data Distribution Service for Real-Time Systems Specification*, version 1.2, January 2007
8. A. Ledeczi *et al.*, "Composing Domain-specific Design Environments," *Computer*, vol. 34, no. 11, pp. 44–51, 2001.
9. M. Xiong, *et al*., "Evaluating Technologies for Tactical Information Management in Net-Centric Systems," in *Proceedings of the Defense Transformation and Net-Centric Systems conference*, Orlando, Florida, Apr. 2007.
10. A. Kavimandan and A. Gokhale, "Automated Middleware QoS Configuration Techniques using Model Transformations," in *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, St. Louis, MO, USA, Apr. 2008, pp. 93–102.
11. J. von Pilgrim, "Measuring the Level of Abstraction and Detail of Models in the Context of MDD," in *Second International Workshop on Model Size Metrics*, October 2007, pp. 10–17.

Explanation of Relevance for Model-Driven Domain Analysis and Software Development: Architectures and Functions

This chapter proposal is relevant to the book in the area of Contributions for Enterprise Computing: Model-driven Distributed Systems. This chapter shows how a DSML is used to address the challenges of QoS configuration development for QoS-enabled middleware for distributed systems. Moreover, this chapter quantitatively shows how productivity is increased when generating implementation artifacts.

Bios

Joe Hoffert is a Ph.D. student in the Department of Electrical Engineering and Computer Science at Vanderbilt University. His research focuses on QoS support for the infrastructure of the Global Information Grid. He previously worked for Boeing in the area of model-based integration of embedded systems. He received his B.A. in Math/C.S. from Mount Vernon Nazarene College (OH) and his M.S. in C.S. from the University of Cincinnati (OH).

Dr. Douglas C. Schmidt is a Professor of Computer Science and Associate Chair of the Computer Science and Engineering program at Vanderbilt University. He has published 9 books and over 400 papers that cover a range of topics, including patterns, optimization techniques, and empirical analyses of software frameworks and domain-specific modeling environments that facilitate the development of distributed real-time and embedded (DRE) middleware and applications. Dr. Schmidt has over fifteen years of experience leading the development of ACE, TAO, CIAO, and CoSMIC, which are open-source middleware frameworks and model-driven tools that implement patterns and product-line architectures for high-performance DRE systems.