Productivity Analysis for the Distributed QoS Modeling Language

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

***Abstract*—Model-driven engineering (MDE), in general, and Domain-Specific Languages (DSLs), in particular, are increasingly being used to manage the complexity of developing applications in various domains. Although many DSL benefits are qualitative, there is a need to quantitatively demonstrate the benefits of DSLs to simplify comparison and evaluation. This paper describes how we conducted productivity analysis for the Distributed Quality-of-Service (QoS) Modeling Language (DQML). Our analysis shows (1) the significant productivity gain using DQML compared with alternative methods when configuring application entities and (2) the viability of quantitative productivity metrics for DSLs.**

***Index Terms*—Model-Integrated Computing, Productivity Analysis, DSL, GME, DQML, QoS Configuration**

I. Introduction

MODEL-DRIVEN Engineering (MDE) helps address the problems of designing, implementing, and integrating 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. There is a lack of documented quantitative benefits for DSLs, however, that show how (1) developers are more productive using MDE tools and (2) development using DSLs yields fewer bugs.

Conventional techniques for quantifying the benefits of DSLs, such as comparing elapsed development time for a domain expert with and without the use of the DSL [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 also often require the analysis of domain experts who are unavailable in many production systems.

Even though DSL 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, we present a lightweight approach to quantitatively evaluating DSLs via *productivity analysis*, which measures how productive developers are and quantitatively explores factors that influence productivity [2], [3].

While there has been much prior work on domain-specific technologies, less attention has focused on quantitative productivity metrics for DSLs. 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 DSL 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 DSL.

This paper focuses on applying quantitative productivity measurement on a case study of the *Distributed QoS Modeling Language* (DQML), which is a DSL 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.

II. Distributed QoS Modeling Language

The *Distributed QoS Modeling Language* (DQML) is a DSL 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) (a pub-sub middleware standard defined by the Object Management Group and summarized in Sidebar 1), though the approach can be applied to other pub-sub technologies. DQML has been developed using the Generic Modeling Environment [7] (GME) which is a meta-programmable environment for developing DSLs. This section provides an overview of DQML's structure and functionality.

A. 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 DDS entity types that can have QoS policies

Sidebar 1: Data Distribution Service (DDS) Overview

DDS (www.omg.org/dds) defines a standard 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.

Key DCPS entities include *topics*, which describe the type and structure of the data to read or write; *data readers*, which subscribe to the data of particular topics; and *data writers*, which publish data for particular topics. Various properties of these entities can be configured using combinations of the 22 QoS policies. In addition, *publishers* manage one or more data writers while *subscribers* manage one or more data readers.

DDS provides a rich set of QoS policies as indicated by the number of QoS policies available. Each QoS policy has $\sim$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 DCPS entities, as well as which combinations of QoS policy values are semantically compatible.

associated with them. Along with the entities described in Sidebar 1, the metamodel also includes support for *domain participants*, which create DDS entities within a particular domain, and *domain participant factories*, which are used to generate domain participants.

DQML's metamodel supports associations between applicable DDS entities, and between entities and applicable QoS policies. Associations between DDS entities are restricted, *i.e.*, not all DDS entities can be associated with any other type of entity (*e.g.*, an association between a subscriber and a data writer is invalid since subscribers manage data readers rather than data writers). DQML enforces the validity of associations between DDS entities.

Likewise, the metamodel supports associations between entities and QoS policies. Once again, associations between entities and QoS policies are restricted, *e.g.*, a time-based filter QoS policy can *only* be associated with a data reader since only the data reader knows if data is being received too quickly. The DQML metamodel enforces the validity of associations between entities and QoS policies.

The DDS specification constrains associations between entities and QoS policies with respect to *compatibility* and *consistency*. Compatibility is relevant to a common type of QoS policy associated with multiple entities (*e.g.*, a reliability QoS policy associated with a data reader and a reliability QoS policy associated with a data writer). Consistency is relevant to multiple QoS policies associated with a single DDS entity (*e.g.* a deadline QoS policy and a time-based filter QoS policy associated with the same data reader).

Constraint definitions in the DQML metamodel enforce compatibility and consistency within a QoS configuration model. These constraints are defined using the *Object Constraint Language* (OCL) [8]. Compatibility and consistency constraint checking can be initiated by DQML users during application development.

B. 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 an arbitrary number of DDS QoS policy instances (*e.g.*, any number of 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.

DQML allows users to generate associations between the DDS entities, as well as between entities and QoS policies. It ensures that users only specify valid associations, *i.e.*, where it is valid to associate two particular types of entities or associate a particular DDS entity with a particular type of QoS policy. DQML allows the association of the same QoS policy instance with more than one entity, *i.e.*, the entities "share" a common QoS policy. This DQML feature enforces correct configurations when identical QoS policy instances are needed to ensure a valid configuration that is also designed as intended (*e.g.*, associating a data reader and data writer to the same reliability QoS policy instance ensures that both entities will be configured with the same reliability settings).

DQML checks for compatible and consistent QoS policy configurations. It reports any violations along with detailed information to aid users in resolving the violations. When no QoS parameters values are specified by the user, DQML uses default QoS parameter values to determine QoS compatibility and consistency. Moreover, DQML will check parameter values for compatibility and consistency if the values are defaults, explicitly set by the user, or a combination of both. Additionally, DQML supports transformation from configuration design to implementation via application-specific interpreters which is detailed in Section IV-B.

III. DQML Case Study: DDS Benchmarking Environment (DBE)

At least five different implementations of DDS are available, each with its own set of strengths and market discriminators. 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.

Since DDS has a large QoS configuration space (as outlined in Sidebar 1) there is an exponential number of testing configurations where QoS parameters can vary in several orthogonal

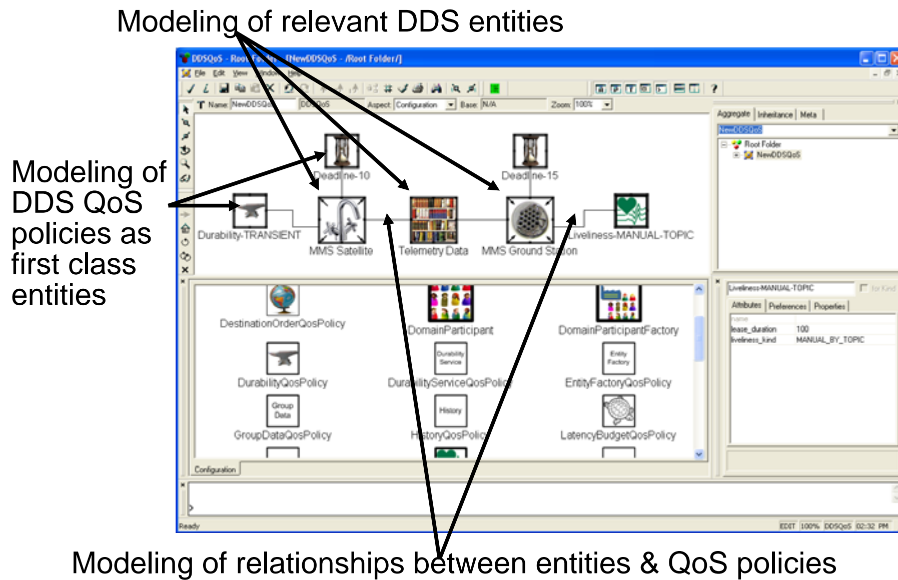

Fig. 1: **The Distributed QoS Modeling Language (DQML)**

dimensions. For example, evaluation scenarios can involve any combination of the following subset of QoS policies:

1) **Durability** to manage data for late arriving subscribers,
2) **Time-based Filter** to provide inter-arrival data spacing, *e.g.*, when a fast publisher of data overwhelms a slow subscriber,
3) **Reliability** to reliably deliver data or to remove reliability overhead and jitter for real-time applications,
4) **Ownership** to provide failover capability,
5) **Resource Limits** to provision data resources,
6) **Transport Priority** to prioritize the transfer of data,
7) **Liveliness** to determine liveness of an application, system, or subsystem,
8) **Presentation** to order data, *e.g.*, when a causal ordering of data needs to be preserved, and
9) **Deadline** to support the timeliness of data.

Manually performing evaluations for each (1) QoS configuration, (2) DDS implementation, and (3) 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.

The *DDS Benchmarking Environment* (DBE) tool suite was developed to examine and evaluate the QoS of DDS implementations [9]. DBE is an open-source framework for automating and managing the complexity of evaluating DDS implementations with various QoS configurations. DBE consists of (1) a repository containing scripts, configuration files, test ids, and test results, (2) a hierarchy of Perl scripts to automate evaluation setup and execution, and (3) a shared C++ library for collecting results and generating statistics.

The architecture of DBE supports three levels of execution to enhance portability, performance, and flexibility, while minimizing overhead. The top level provides the user interface, the middle level manages the platform or node, and the bottom level constitutes the executables (*e.g.*, publishers and subscribers for each DDS implementation).

DBE deploys a QoS policy configuration file for each data reader and data writer. As shown in Figure 2, 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.

We use DBE as a case study in this paper 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 a 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-

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

Fig. 2: **Example Portion of a DBE QoS Settings File**

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 we are focusing 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 and C++. Manual creation can incur the same accidental complexities as the DBE case study without the integration of MDE tools like DQML.

IV. DSL Productivity Analysis

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

A. Productivity Analysis Approach

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

1) **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 DSL,
2) **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 DSL,
3) **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 DSL,
4) **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 DSL, and
5) **Solution exploration**, comparing the number of viable solutions considered for a particular problem in a set period of time using the DSL as compared to traditional methods or other DSLs.

This article 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.

B. DQML Productivity Analysis

Below we analyze the effect on productivity and the break-even point of using DQML as opposed to manual implementations of QoS policy configurations for DBE. Although configurations can be *designed* using various methods as outlined in previous work [12], manual *implementation* of configurations is applicable to these other design solutions since these solutions provide no guidance for implementation.

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 in Section III) since the files are uniquely and descriptively named.

1) Scope: DBE currently 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 I, 15 QoS policies with a total of 25 parameters can be associated with a single data writer.

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

TABLE I: **DDS QoS Policies for data writers**

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

TABLE II: **DDS QoS Policies for data readers**

Likewise, Table II shows 12 QoS policies with a total of 17 parameters can be associated with a single data reader. Within the context of DBE, therefore, the total number of relevant QoS parameters is **17 + 25 = 42**. 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.

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 DSL interpreter only need to modify and add a small subset of the framework code to traverse and appropriately process the particular DSL 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.

We developed the DQML-specific code for the DBE interpreter utilizing ~160 C++ statements within the BON2 framework. 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.

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 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, and (2) verifying the number, type, and valid values for the parameters of the applicable QoS policies. When implementing a non-trivial QoS policy configuration, therefore, development of the DQML-specific C++ code for the 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. Section IV-B3 provides additional detail into what can be considered a non-trivial QoS policy configuration.

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 already exists. The development effort metrics of 160 C++ statements are included *only* to be used in comparing manually implemented QoS policy configurations.

3) Analysis: 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 Tables I and II for data readers and data writers, Figure 3 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 = 160 QoS policy parameter settings).

Table III also shows productivity gains as a percentage for various numbers of data readers and data writers. The percentage gains are calculated via 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. 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 Params	Productivity Gain
5	210	31%
10	420	163%
15	630	294%
20	840	425%
25	1050	556%

TABLE III: **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

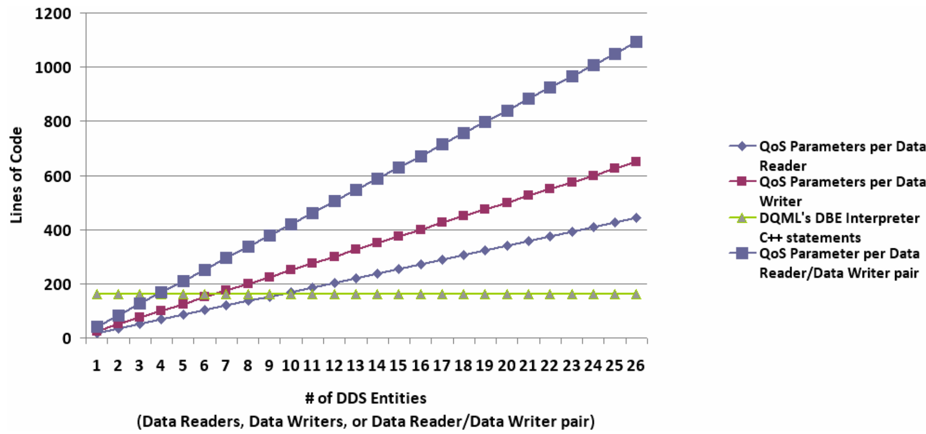

Fig. 3: **Metrics for Manual Configuration vs. DQML's Interpreter**

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.

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 *42,000* (*i.e.*, 17 * 1000 + 25 * 1000). 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 substantially (*i.e.*, 42,000 / 160 = 262.5).

V. Concluding Remarks

Although MDE and DSLs have become increasingly popular, quantitative evidence is needed to support the quantitative evaluation of DSLs. This paper described various approaches to quantitatively evaluating DSLs 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 applying productivity analysis to DQML:

1) **Trade-offs and the break-even point for DSLs must be clearly understood and communicated.** There are pros and cons to any technical approach including DSLs. The use of DSLs may not be appropriate for every case and these cases must be evaluated to provide balanced and objective analysis.
2) **The context for DSL productivity analysis needs to be well defined.** Broad generalizations of a DSL 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.
3) **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 DSL 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, J. Ye, R. Shapiro, S. Neema, N. Mahadevan, S. Abdelwahed, M. Koets, and D. Varner, "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, M. Shepperd, B. Kitchenham, and P. Forselius, "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, D. C. Schmidt, Z. Molnar, and A. Ledeczi, "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] Ákos Lédeczi, Árpád Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," *Computer*, vol. 34, no. 11, pp. 44–51, 2001.

[8] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[9] M. Xiong, J. Parsons, J. Edmondson, H. Nguyen, and D. C. Schmidt, "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.

[12] J. Hoffert, D. Schmidt, and A. Gokhale, "A QoS Policy Configuration Modeling Language for Publish/Subscribe Middleware Platforms," in *Proceedings of International Conference on Distributed Event-Based Systems (DEBS)*, Toronto, Canada, Jun. 2007, pp. 140–145.

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.

Prof. Aniruddha S. Gokhale is an Assistant Professor of Computer Science and Engineering in the Dept. of Electrical Engineering and Computer Science at Vanderbilt University, Nashville, TN, USA. He received his BE (Computer Eng) from Pune University in 1989; MS (Computer Science) from Arizona State University, Tempe, AZ in 1992; and D.Sc (Computer Science) from Washington University, St. Louis, MO in 1998. Prior to joining Vanderbilt, he was a Member of Technical Staff at Bell Labs, Lucent Technologies in New Jersey. Dr.Gokhale is a member of IEEE and ACM. Dr. Gokhale's research combines model-driven engineering and middleware for distributed, real-time and embedded systems.