Model-driven Generative Framework for Automated OMG DDS Performance Testing in the Cloud *

Kyoungho An, Takayuki Kuroda
Aniruddha Gokhale
ISIS, Vanderbilt University, Nashville, TN 37235, USA
{kyoungho, kuroda, gokhale}@isis.vanderbilt.edu

Sumant Tambe and Andrea Sorbini
RTI, Sunnyvale, CA, USA
{sumant, asorbini}@rti.com

Abstract

The Object Management Group's (OMG) Data Distribution Service (DDS) provides many configurable policies which determine end-to-end quality of service (QoS) of applications. It is challenging to predict the system's performance in terms of latencies, throughput, and resource usage because diverse combinations of QoS configurations influence QoS of applications in different ways. To overcome this problem, design-time formal methods have been applied with mixed success, but lack of sufficient accuracy in prediction, tool support, and understanding of formalism has prevented wider adoption of the formal techniques. A promising approach to address this challenge is to emulate system behavior and gather data on the QoS parameters of interest by experimentation. To realize this approach, which is preferred over formal methods due to their limitations in accurately predicting QoS, we have developed a model-based automatic performance testing framework with generative capabilities to reduce manual efforts in generating a large number of relevant QoS configurations that can be deployed and tested on a cloud platform. This paper describes our initial efforts in developing and using this technology.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Testing tools

General Terms Design, Performance

Keywords Model-driven Engineering, Generative Programming, Publish/Subscribe, Performance Testing

1. Introduction

The OMG Data Distribution Service (DDS) [4] is a general-purpose middleware supporting real-time publish/subscribe semantics [1] for mission-critical applications. Specifically, the OMG DDS supports real-time, topic-based, data-centric, scalable, deterministic and anonymous pub/sub interaction semantics for large-scale distributed applications. To support the quality of service (QoS) requirements of a broad spectrum of application domains, OMG DDS supports many QoS configuration policies (in the form of configuration parameters) that when used in different combinations determine the delivered end-to-end QoS properties.

The OMG DDS specification supports a total of 22 individual QoS policies that can be combined in a variety of ways that influences the delivered QoS in different ways. Each QoS policy may have multiple attributes associated with it, such as the data topic of interest, data filter criteria, and the maximum number of data messages to store when transmitting data. Moreover, each attribute can be assigned one of a range of values, such as the legal set of topics, a range of integers for the maximum number of data messages stored for transmission, or the set of criteria used for filtering.

An important consideration with DDS QoS policies is that not all QoS policies can be combined with each other since certain combinations tend to be incompatible with each other. Similarly, the values chosen for specific QoS policies may tend to become inconsistent when combined. Both the incompatibility and inconsistency issues pose significant challenges for DDS application developers who must ensure that their deployed applications have compatible and consistent QoS configuration policies. Our prior work [2] utilized model-driven engineering (MDE) techniques to pinpoint existence of such errors at design-time.

Addressing these accidental challenges alone is not sufficient, however, towards realizing high confidence DDS-based applications. Every individual QoS policy tends to impact the end-to-end performance and behavior of the application in specific ways. When these QoS policies are combined in various combinations, it is hard to predict the outcome on QoS of combining these policies. Such a problem is faced not just by application developers but also by the OMG DDS vendors themselves, who must have an indepth knowledge of how various combinations of configuration parameters interact, and to address issues raised by their customers.

It is not possible to expect an application developer or a vendor to manually write test cases that can test every QoS policy and all possible combinations of these QoS policies (along with their values), not to mention that they must also ensure that these combinations are valid. Even if one were to develop these large number of tests, executing them sequentially is time consuming, which impacts both the application developers who aim at getting their applications to market rapidly and vendors who must address customer problems in a timely manner.

To address the combinatorial testing problem and limitations of sequential testing, this paper presents AUTOMATIC (AUTOmated Middleware Analysis and Testing In the Cloud), which is a framework we have developed that combines MDE techniques with multiple stages of generative capabilities. Specifically, AUTOMATIC provides a domain-specific modeling language that developers use

* This work was supported in part by NSF CAREER Award CNS 0845789. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[Copyright notice will appear here once 'preprint' option is removed.]

to model their applications and QoS policies of interest. Generative tools synthesize essentially a product line of test cases, each testing different QoS policies for the same publish/subscribe business logic. A second set of generators synthesize cloud-based deployment logic. Finally, a testing framework automates the testing of the generated test cases in parallel in the cloud. Although a related effort called Expertus [3] uses aspect oriented weaving techniques for code generation and automated testing of applications for performance in the cloud, this effort does not address the QoS configuration combinations and their impact on performance that we address in this paper.

The rest of the paper is organized as follows: Section 2 describes the overall approach providing brief details of each stage in our approach; Section 3 provides initial insights gained in validating our solution; and finally Section 4 offers concluding remarks alluding to work needed to make the work robust and complete.

2. Design and Implementation of AUTOMATIC

Figure 1 describes the overall architecture and workflow of our automated performance testing framework called AUTOMATIC. AUTOMATIC comprises three activity domains: User, Test Automation System, and Cloud Infrastructure. The Modeling and Monitoring functions included in the User domain should be conducted by a user who prototypes DDS applications and performs performance testing of the applications. In the Test Automation System domain, Test Planning and Test Deployment functions are carried out by predefined tools in our framework. When the Test Planning is completed and ready to be deployed in a testing infrastructure, a test environment is generated for our cloud infrastructure to emulate application testing.

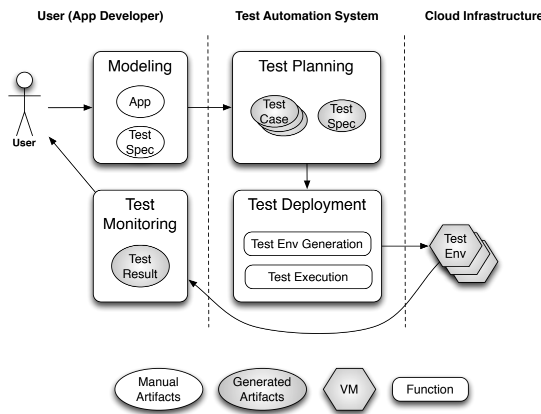

Figure 1. Framework Architecture

The rest of this section describes each activity in detail including the performance monitoring capability.

2.1 Domain-Specific Modeling Language

We developed a DSML using the Generic Modeling Environment (GME) (`www.isis.vanderbilt.edu/projects/GME`) that supports modeling a DDS application for emulation and testing its performance for various combinations of DDS QoS policies. GME provides a meta-modeling environment to develop DSMLs for specific domains. Our meta-model includes modeling elements for all OMG DDS entities including Domain, Topic, Publisher, Subscriber, DataWriter, DataReader, QoS, and their connections. In DDS applications, a scope or operating region of an application is determined by the Domain, and applications are isolated by different Domain IDs. DDS applications publish or subscribe via Data Writers and Data Readers through associated Topics, and therefore in the meta-model the Topic and Type elements are contained in the Domain element and Topics and Types in the Domain are accessible by DataWriter and DataReader entities running in the same Domain. Moreover, the Domain contains a Participant element which is a concept to represent a processing unit for publishing or subscribing or both. Lastly, the modeling capability to configure QoS policies for DDS entities is contained in the Domain element. Data communications between Participants are differentiated and identified by a Topic, so a TopicConnection element is required in the Domain model to be used by Participants.

Figure 2 shows an example testing application defined with our modeling language. The application's topology is shown in the bottom of the figure. This example application examines the throughput of the application publishing data from a Participant containing a DataWriter to a Participant involving each DataReader. Each DataWriter and DataReader are placed under the Participant element and behaves as a communicating port between Participants.

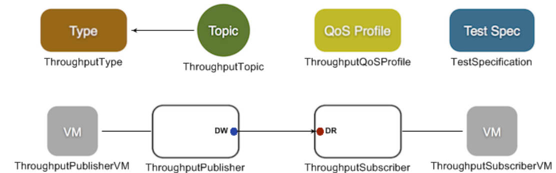

Figure 2. Example Domain-Specific Model of DDS Throughput Testing Application

The DDS Participants are deployed in virtual machines (VMs) for testing and each Participant in the model are connected to a VM element based on the deployment decision by users. In this example, each Participant is deployed in each different VM. The deployment plan (mapping of Participants and VMs) can be flexibly altered by users in the modeling language if users like to test with different deployment plans. Users can emulate their applications by setting analogous hardware specifications to find similar performance results in actually deployed environment.

Communicating DataWriters and DataReaders are connected with directed lines which indicate physical communications defined by a Topic. A Topic is shown in the top of this example model. If a name of a line is the same as the name of a Topic, it means DataWriters and DataReaders connected with the link communicate data by the Topic. Each data type of a Topic is determined by a struct like Type.

In the QoS Profile element, QoS policies used by DataWriters and DataReaders are contained. For example, Reliability QoS has two kinds of policies to determine the level of reliability: RELIABILITY and BEST_EFFORT. History QoS also has two kinds of policies to set the number of history samples in a entity's cache: KEEP_ALL and KEEP_LAST. Some QoS policies need to set as numeric values such as history depth in History QoS.

Finally, the configurable parameters are set in the TestSpec element. In this element, test related information such as running duration of the test, and the number of test cases concurrently running is configured. A deployment tool uses this information to decide the number VMs in a test set and schedule the test operations.

2.2 Test Plan Generation

The Test Planning function traverses the modeled elements in a model instance via a model interpreter to generate executable applications and related test specification files.

Figure 3 shows a XML-based DDS application tree model transformed by the model interpreter based on Figure 2. Because the aim of our automatic testing framework is to analyze application performance by varying QoS configuration, elements under the QoS Library are categorized into variable elements and the rest of elements fall into the common elements category. This approach is conducive to using generative programming to realize a product line of test cases. The QoS Library embodies QoS elements for DataWriters and DataReaders. To demonstrate our framework with a simple example, we varied only the Reliability QoS. In this example, both DataWriter QoS and DataReader QoS have Reliability QoS. BEST_EFFORT or RELIABILITY can be selected as a kind of the Reliability QoS.

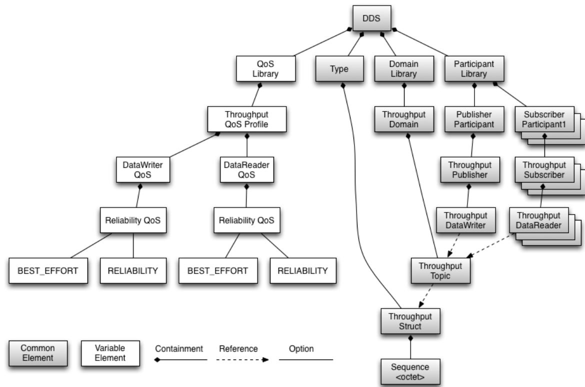

Figure 3. XML-based DDS Application Tree

The following procedure is used to form a tree shown in Figure 4 for all possible combinations of QoS configurations defined in the QoS Library. In the example, four test cases can be generated as each DataWriter and DataReader QoS has Reliability QoS that can choose from BEST_EFFORT and RELIABLE. Once the combination tree for variable elements is complete, the combination tree is traversed with depth-first search to create trees for variables elements actually used by the applications for testing.

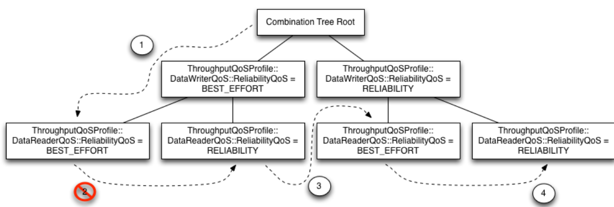

Figure 4. Variable Element Combination Tree

As a final outcome, four trees for variable elements are created as shown in Figure 5. The tree numbered 2 is discarded by the interpreter because the QoS configurations are not compatible. The reason is that if the DataWriter's Reliability QoS is BEST_EFFORT and DataReader's Reliability QoS is RELIABLE, then no communication between them is feasible according to the DDS specification.

We checked for all compatibility and inconsistency violations in the model interpreter though this task can be accomplished using the Object Constraint Language in the model itself as shown by our prior work [2] or the runtime environment may also be able to flag these cases as errors. As the final step, the trees for variable elements are combined with the tree for common elements introduced in Figure 3, and the executable applications are generated.

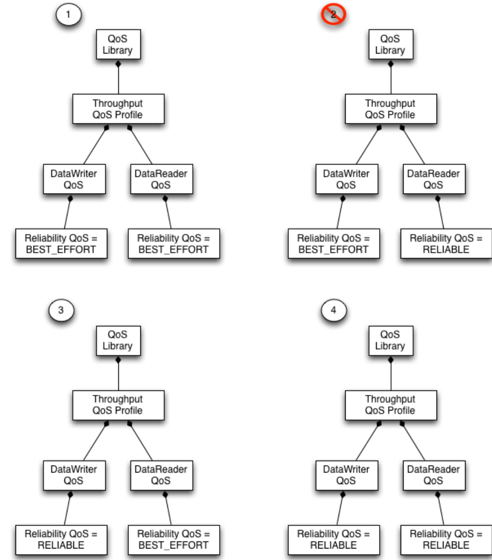

Figure 5. Variable Element Tree

2.3 Test Deployment

To deploy the XML-based DDS testing applications in a cloud-based testing infrastructure, specifications related to the deployment are also generated by the model interpreter. The specifications are composed of three parts: Test Specification, VM Specification, and Application Specification. The Test Specification describes the environment including a reference to the VM Specification, concurrency level, duration for test execution, publication period of publishers. Each test case is defined with an assigned ID and a referring specification file. The referred specification files have information about application's topology and the execution command.

The VM Specification example describes required VMs for testing and information of VMs such as VM instance type and image. These specifications are fed into our deployment tool. VM instance types indicate specifications of VM such as the number of virtual CPUs, memory size, and storage capacity. According to the user-selected VM image and VM instance type, the Test Env Generation function deploys a proper VM in a cloud infrastructure. When the VM has booted up, a SSH connection is established and a test case application is sent to the VM over the SSH connection by the Test Execution function.

We implemented our deployment tool in Python 2.7 for the Test Deployment function. Our private cloud for testing adopted OpenStack as a cloud operating system, and the Python Boto library is exploited to control cloud resources via Amazon AWS APIs. The generated XML-specified application that is moved to the deployed VM is subsequently executed on that VM using a tool provided by RTI called the RTI Prototyper (`http://community.rti.com/content/page/download-prototyper`).

2.4 Test Monitoring

We employed another product from RTI called the RTI Monitor to detect DDS applications' performance while it is executing on the VM. The RTI Monitor is a tool to visualize monitoring data of applications. The RTI Monitor helps users to understand their systems easily via graphical interfaces and to verify behaviors of entities as expected. Moreover, it comes to the aid of improving performance throughput provided statistics such as CPU and memory usage, and

throughput. The experimental results illustrated in Section 3 were collected using this tool.

3. Technology Validation

Our efforts at validating the claims in AUTOMATIC thus far have focused on a scenario where an application developer seeks to make appropriate tradeoffs trying to balance the conflicting requirements of reliability and timeliness. To that end, the experiment evaluates performance of an example DDS application by combining the RELIABILITY, HISTORY and DEADLINE QoS policies. In this experiment, DDS applications use core libraries of RTI Connext DDS 5.0 (which is an implementation of OMG DDS) and executable scripts provided with RTI Connext Prototyper 5.0. Our OpenStack-based cloud testbed employs KVM as a virtual machine (VM) hypervisor. Each VM machine type used in this experiment consists of 1 virtual CPU and 512 MB memory.

In our example, the publisher periodically publishes a topic containing octet sequence typed data of 64K bytes to the subscriber. We chose a large packet size in the hope of congesting the network. The publishing period is decided by the DEADLINE QoS setting and was fixed at 1 millisecond. The purpose of this experiment is to understand deadline miss rate for different RELIABILITY QoS configurations. The HISTORY setting was KEEP_ALL, which means the publisher and subscriber hold on to all the data samples so they can be used for retransmissions when complete reliability is desired. The RELIABILITY setting is varied between RELIABLE (for eventual consistency) versus BEST_EFFORT (where no attempt is made to retry transmissions when samples are lost). The generated test cases are shown in Figure 5.

Figure 6 shows deadline miss counts of DataReader's (an entity on the subscriber side) in the test cases. If a sample is not arrived in a DataReader within 1 millisecond, it is counted as a missed deadline. Each test case runs for 4 minutes and values are monitored every 5 seconds. The X axis indicates time and Y axis presents deadline missed samples for 5 seconds.

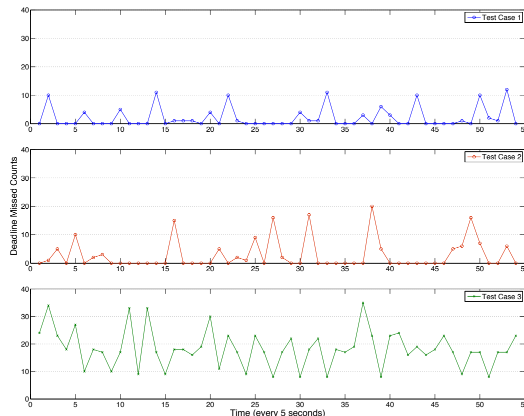

Figure 6. Deadline Miss Counts for Different Reliability QoS Settings

In test case 1, most samples do not miss the deadline and the range of the samples spans up to 10 as maximum. If Reliability QoS is set to BEST_EFFORT, a DataWriter keeps publishing data regardless of the status of a DataReader and therefore it is beneficial to be used for applications demanding low latency. In a congested network environment, it would possibly lose samples, however, since our test network not congested, there were no lost samples.

In test case 3, deadline miss counts are monitored from 9 to 35 where they keep occurring during the entire testing period. If the data cache of a DataWriter with the RELIABLE Reliability QoS is filled with unacknowledged samples, the DataWriter's write operation is blocked for a while to control the sending rate to avoid congestion which increases the latency of samples delivered. Accordingly, high latency causes deadline miss counts on the DataReader side. However, samples can reliably arrive at the DataReader due to the middleware supporting the retransmissions.

4. Concluding Remarks

Modern middleware, such as the OMG Data Distribution Service (DDS), provide substantial flexibility to applications by virtue of supporting a large number of configuration options. These configuration options when combined in different ways can lead to vastly different performance and behavioral characteristics for the applications. Although some intuition is always available on the potential impact of individual configurations, and some guidelines do emerge after a few years of experience using multiple configurations on real applications (*e.g.*, `community.rti.com/best-practices`), application developers continue to face numerous challenges deciding the right combinations of options they must use for their application for the chosen deployment environments. It is infeasible for developers to manually create and test each possible scenario to understand the impact of the configuration options.

To address these challenges in the context of OMG DDS middleware, this paper combines model-driven engineering (MDE) and generative programming techniques to provide a tool called AUTOMATIC (AUTOmated Middleware Analysis and Testing In the Cloud). MDE helps application developers with intuitive abstractions to rapidly describe their scenarios. Generative programming is needed since the test cases that combine configuration options can be considered a product line where the DDS application business logic remains common while the configurations can vary. Deployment and testing in the cloud is chosen as an approach because of its elastic nature where we can automate the parallel execution and collection of test statistics for a large number of generated tests from our tooling. Although the presented technology is showcased for the OMG DDS middleware, the principles behind AUTOMATIC are applicable to other middleware. Moreover, our technology has significant practical utility to both application developers and middleware vendors.

The presented work illustrates the feasibility of such an idea. Our ongoing work is focusing on making AUTOMATIC complete and robust for OMG DDS, and test it on a large number of deployment scenarios. Future work is also looking into generating application business logic. Current artifacts in AUTOMATIC are available for download from `www.dre.vanderbilt.edu/~kyoungho/AUTOMATIC/`.

References

[1] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computer Survey*, 35:114–131, June 2003.

[2] Joe Hoffert, Douglas Schmidt, and Aniruddha Gokhale. A QoS Policy Configuration Modeling Language for Publish/Subscribe Middleware Platforms. In *Proceedings of International Conference on Distributed Event-Based Systems (DEBS)*, pages 140–145, Toronto, Canada, June 2007.

[3] D. Jayasinghe, G. Swint, S. Malkowski, J. Li, Qingyang Wang, Junhee Park, and C. Pu. Expertus: A Generator Approach to Automate Performance Testing in IaaS Clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 115–122, 2012.

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