Tools for Continuously Evaluating Distributed System Qualities

James H. Hill
Indiana University/Purdue University at Indianapolis
Indianapolis, IN, USA
hillj@cs.iupui.edu

Douglas C. Schmidt, James R. Edmondson, and Aniruddha Gokhale
Vanderbilt University
Nashville, TN, USA
{d.schmidt, james.r.edmondson, a.gokhale}@vanderbilt.edu

Abstract

The end-to-end evaluation of quality-of-service (QoS) properties (e.g., performance, reliability, and security) for distributed systems has historically occurred late in the software lifecycle. As a result, many design flaws that affect QoS are not found and fixed in a timely and cost-effective manner. This article shows how model-driven engineering—particularly domain-specific modeling languages coupled with system execution modeling tools—can enable agile development of distributed systems and facilitate continuous system integration testing to improve quality assurance of QoS properties throughout the software lifecycle.

Keywords. agile techniques, continuous system integration, distributed systems, domain-specific modeling languages, model-driven engineering, system execution modeling tools

1 Introduction

Current trends and challenges. Service-oriented middleware [7] is increasingly used to develop distributed systems. This middleware raises the level of abstraction for software so that distributed system developers can focus more on application-level concerns (*e.g.*, the "business logic") rather than wrestling with infrastructure-level concerns (*e.g.*, adaption, context-awareness, and lifecycle management). Service-oriented middleware also promotes reuse of business-logic and services across heterogeneous application domains, which facilitates the development of larger and more complex systems [2].

As service-oriented distributed systems grow in size and complexity, it becomes harder to ensure that they conform to their specifications throughout the software lifecycle. This difficulty stems in part from the *serialized phasing problem* [8], where application-level entities are developed after infrastructure-level entities. Serialized phasing makes it hard to evaluate end-to-end functional and quality-of-service (QoS) aspects until late (*e.g.*, at system integration time) in the software lifecycle.

Agile techniques [6] help address functional aspects of serialized phasing by continuously validating software functionality throughout the software lifecycle [9]. For example, test-driven development and continuous integration are agile techniques that validate functional quality by ensuring software behaves as expected throughout its lifecycle. The benefits of using agile techniques to improve QoS assurance of service-oriented distributed systems, however, has not been demonstrated. Developers therefore need new techniques that help alleviate the complexity of serialized phasing and enable evaluation of QoS concerns continuously throughout the software lifecycle.

Promising approach → Agility via model-driven engineering (MDE) techniques. MDE [10] is a promising solution for improving software development of service-oriented distributed systems. MDE techniques, such as domain-specific modeling languages (DSMLs) [4], provide developers with visual representations of abstractions that capture key domain semantics and constraints. DSMLs also provide tools that transform models into concrete artifacts (such as source code or configuration files) that are tedious and error-prone to create manually using third-generation languages or not available early enough in the software lifecycle to evaluate end-to-end QoS properties properly.

This article presents our approach for using DSMLs to realize agile techniques for evaluating service-oriented distributed system QoS continuously throughout the software lifecycle. Our approach is based on *system execution mod-*

eling methods and tools [11] that enable developers to conduct the following agile quality assurance process:

1. Rapidly model *behavior and workload* of the distributed system being developed, independent of its programming language or *target environment* (*e.g.*, the underlying networks, operating system(s), and middleware platform(s);
2. Synthesize a customized *test system* from models, including representative source code for the behavior and workload models and project/workspace files necessary to build the test system in its target environment;
3. Execute the synthesized test system on a representative *target environment testbed* to produce realistic empirical results at scale; and
4. Analyze the test system's QoS in the context of domain-specific constraints (such as scalability or end-to-end response time of synthesized test applications) to identify performance anti-patterns [11], which are common system design mistakes that degrade QoS.

We have realized such agile techniques in an open-source tool called CUTS (`www.cs.iupui.edu/CUTS`), which is a language-, operating system-, and middleware-independent DSML-based system execution modeling tool for service-oriented distributed systems.

The remainder of this article uses CUTS as a case study to qualitatively and quantitatively evaluates how DSML-based system execution modeling tools can support lightweight and adaptable software development and QoS assurance processes. Developers can use these tools to quickly pinpoint performance bottlenecks and other QoS concerns throughout the lifecycle, rather than wrestling with low-level test implementations written in third-generation languages.

2 Assuring QoS in Service-Oriented Distributed Systems

This section describes how the following CUTS DSMLs provide key capabilities needed to assure QoS in service-oriented distributed systems:

- the *Component Behavior Modeling Language* (CBML) DSML, which is used to model component behavior (see Section 2.1);
- the *Workload Modeling Language* (WML) DSML, which is used to model component workload (see Section 2.2); and
- the *Understanding Non-functional Intentions via Testing and Experimentation* (UNITE) DSML, which is use to specify QoS unit tests for performance analysis in distributed systems (see Section 2.3).

Figure 1 shows how CUTS's DSMLs map to the agile QoS assurance process described in Section 1. To focus the discussion, we describe how these CUTS DSMLs and their agile techniques are applied to the QED project described in Sidebar 1.

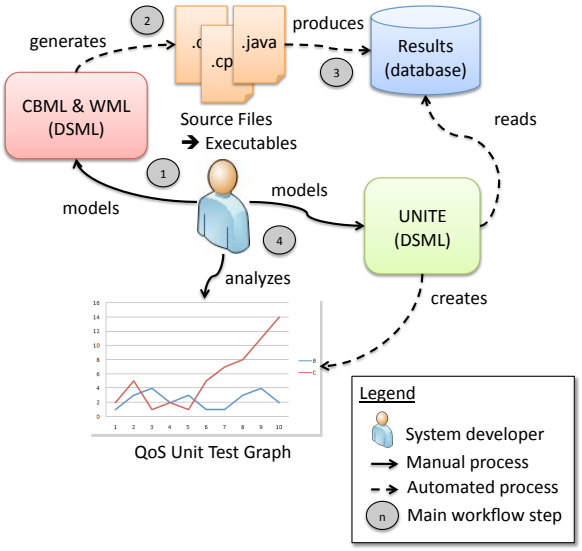

Figure 1. CUTS workflow and domain-specific modeling languages.

Sidebar 1: Overview of the QED Project

The *QoS-Enabled Dissemination* (QED) project [5] is a large-scale, multi-team collaborative project that focuses on QoS-enabled service-oriented infrastructure and applications in the *Global Information Grid* (GIG) [1]. The GIG middleware must provide dependable and timely communication to applications and end-user scenarios that operate within dynamically changing conditions and environments, such as wireless ad-hoc networks and/or bandwidth and resource-constrained situations. The QED middleware enhancements for the GIG therefore provide timely delivery of information needed by users in mobile scenarios, tailoring and prioritizing information based on mission needs and importance, and operating in a manner that is robust to failures and intermittent communications.

2.1 Capability 1: Capturing Behavior and Workload

Service-oriented distributed systems are typically reactive and respond to inputs, such as events or remote method

invocations. The behavior and workload for such systems is analogous to a sequence of actions that cause side-effects, such as changing the value of a variable, querying a database, or generating output events. DSML-based system execution modeling tools must therefore capture these properties to provide a lightweight adaptive process to model behavior and workload rapidly. These tools must also use intuitive domain-specific abstractions rather than manually implementing distributed systems using tedious and error-prone third-generation programming languages.

We have realized this capability in CUTS by providing two DSMLs called the *Component Behavior Modeling Language (CBML)* and the *Workload Modeling Language (WML)* (step 1 in Figure 1) that simplify modeling behavior and workload, respectively. Developers use CBML to define the behavior of a component, which is software that encapsulates common services for individual entities of a service-oriented distributed system using action-to-state sequence diagrams. Likewise, WML is used characterize component workload by parameterizing actions in CBML, *e.g.*, by setting their values.

In CBML, *actions* represent operations (such as an application component receiving/sending an event or querying a database) and *states* represent the current value of the systems variables. Each action can also cause an *effect* that may cause a component's state to change (such as incrementing a component's event counter variable). In WML, *workload generators (workers)* represent objects (such as C++ or Java classes) that perform predefined behavior and *worker actions* represent object methods, *i.e.*, the predefined behavior.

Figure 2 shows a partial behavior and workload model for a multistage workflow application component used to evaluate QED's QoS (see Section 3). In this figure, the behavior and workload of the `receiveTarget` event begins with an *input action* (*i.e.*, the leftmost square). After the initial input action, a sequence of *actions* (such as `logMessage`) and *states* define the behavior. The component has a worker named `logger`, which reports user-defined information about the component in the format described in Section 2.3.

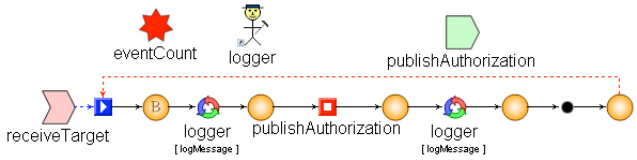

Figure 2. Behavior and workload model for a multistage workflow application component.

Since QED developers model behavior and workload using DSMLs, they can easily adapt their models to test different scenarios, such as changes in QED's specification or evaluating different QoS concerns (see Section 3.1). For example, if QED developers need to add new behavior and workload to log more/less information about the system, updating existing CBML and WML models is straightforward. Moreover, DSMLs shield developers from wrestling with low-level infrastructure details, which can be time-consuming and error-prone.

2.2 Capability 2: Generating Realistic Data

Assuring service-oriented distributed systems QoS requires generating realistic data and results, such as the service time of components, continuously throughout the software lifecycle. DSML-based system execution modeling tools should therefore produce realistic results and feedback throughout the software lifecycle. We have realized this capability in CUTS by leveraging model interpreters to auto-generate *faux* application components based on the constructed behavior and workload models (see Section 2.1). These generated *faux* components conform to the interfaces of the actual components being developed. This conformance enables the incremental replacement of *faux* components as development evolves—thereby enabling continuous system integration and evaluation of QoS throughout the software lifecycle.

After QED developers use CBML and WML to model the behavior and workload of components in the multistage workflow application (step 1 in Figure 1), they use CUTS model interpreters to generate source code customized for their target environment (step 2 in Figure 1). Listing 1 shows a portion of the Java source code for the multistage workflow model in Figure 2.

```
1  void receiveTarget (TargetEvent ev) {
2   try {
3    // effect: update the eventCount
4    ++ eventCount;
5
6    if (eventCount % publishRate == 0) {
7     // generate log message
8     this.logger_.logMessage (LM_INFO
9      instanceName + ": Event " + eventCount +
10     ": Received a TargetMio at " +
11     System.currentTimeMillis ());
12
13    // create new event for publishing
14    JbiEvent <AuthorizationType> ev_1_ =
15     new JbiEvent <AuthorizationType> (
16     AuthorizationType.class);
17
18    ev_1_.setPayload (1024);
19    ev_1_.setMetadata (/* metadata */);
20
21    // publish event to GIG
```

```
22      this.publishAuthorization_.
23        publishData (ev_1_);
24
25      // generate log message
26      this.logger_.logMessage (/* message */);
27    }
28   }
29   catch (Exception e) {
30    e.printStackTrace ();
31   }
32  }
```

Listing 1. Portion of auto-generated source code for a multistage workflow application component.

By using DSML-based system execution modeling tools and model interpreters, QED developers can implement a complete test system and evaluate QoS rapidly. Since CBML and WML are language-, OS-, and middleware-independent, moreover, existing models can scale and adapt (see Section 3.2) to different environments. For example, the multistage workflow application uses the CUTS interpreter for the Java-based QED middleware. If QED developers want to generate source code for different environments (such as Microsoft.NET or the OMG Data Distribution Service), the their existing models can remain unchanged and the model interpreters adapt as needed. This typically involves defining mapping functions from CBML/WML to the target architecture, and manually implementing them in the CUTS model interpreter.

2.3 Capability 3: Collecting and Analyzing Distributed System Data

Collecting and analyzing QoS metrics in service-oriented distributed systems is hard since the data to collect and analyze often changes over time. System structure may also change over the lifecycle (*e.g.*, by altering a component's interface, increasing the number of replicated components, or modifying system deployments by add/removing connections between components), which can affect how data is analyzed. Data collection and analysis techniques must therefore adapt to the volatility in service-oriented distributed systems.

We have realized this capability in CUTS using the *Understanding Non-Functional Intentions via Testing and Experimentation (UNITE)* DSML to analyze end-to-end QoS independent of data and system complexity using system execution traces generated during a test run (step 4 in Figure 1). Developers use UNITE to define QoS unit tests, which include:

- a set of *log formats* that identify data to extract from system execution traces (such as the event count in the log message from the `logMessage` action in Figure 2 and line 8 in Listing 1), which allows UNITE to adapt to many variants of the same log format when extracting data for QoS metrics;
- a set of *causal relations* that specify the order of occurrence for each log format in the system execution trace, which allows UNITE to correlate and evaluate distributed data irrespective of system structure and complexity; and
- a user-defined *evaluation function* f (such as the end-to-end response time of multistage workflow application based on variables in log formats for extracting data), which allows UNITE to adapt evaluation of complex QoS metrics without *a priori* knowledge of data complexity and system complexity.

Developers can also define (1) an *aggregation function*, such as `SUM(`f`)` and `AVG(`f`)`, that combines multiple occurrences of a result and (2) a *grouping criteria* that partitions results into sets before aggregation. By removing the aggregation function, developers can view the data trend of the QoS unit test throughout the system's execution in its target environment.

Listing 1 highlights two different log messages (line 8 and line 26) generated by a multistage workflow application component. Likewise, Figure 3 shows the corresponding UNITE specification for locating the log messages generated in Listing 1, which helps evaluate QED QoS (see Section 3). Equation 1 shows a simple equation for calculating average service time based on the partial specification in Figure 3.

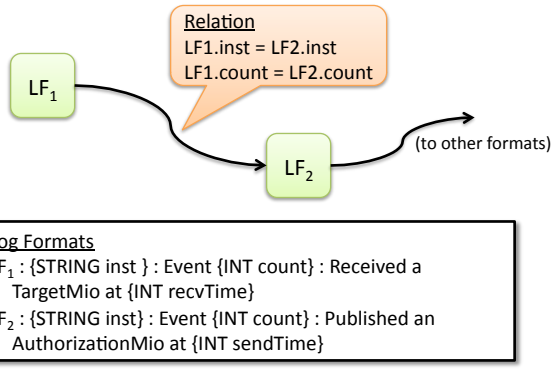

Figure 3. Partial UNITE specification for multistage workflow application QoS unit test.

$$f = AVG(LF_2.sendTime - LF_1.recvTime) \quad (1)$$

UNITE provides QED developers with a lightweight technique to assure QoS by automatically extracting metrics of interest from system traces. Moreover, UNITE does not require QED developers to understand distributed system composition when analyzing extracted data. For example, if QED developers increase the number of components in the multistage workflow application or want to extract more/less data from system traces, UNITE can adapt to such scenarios (see Section 3.2).

3 Evaluating Agile Techniques for QoS Assurance

This section presents the results of an experiment that applied CUTS's DSMLs to the multistage workflow application, which is a representative application that runs atop the GIG middleware, to evaluate the QoS of the QED and GIG middleware. The multistage workflow application consists of six different component stages, each represented by a rectangular object shown in Figure 4. The lines connecting each component represent a communication channel passing through the QED middleware and GIG infrastructure. Each application component contains a CUTS behavior and workload model (see Section 2.1) that stresses different parts of the QED and GIG middleware by increasing network traffic at controlled intervals. Each component also contains actions that log metrics for UNITE (see Section 2.3) to analyze QED and GIG middleware performance. Finally, the experiments described below were run in a representative target environment testbed at ISISlab (`www.isislab.vanderbilt.edu`).

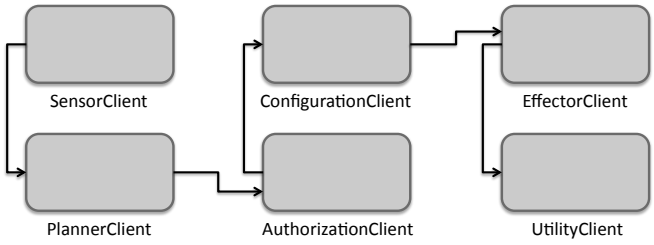

Figure 4. Structural model of the multistage workflow application.

3.1 Baselining the GIG Middleware

The response time of the GIG middleware, *i.e.*, one-way latency for sending/receiving events between 2 components, is important because it helps determine the existing QoS capabilities of the GIG middleware and where the QED middleware can improve QoS relative to the GIG middleware baseline. In [3] we conducted experiments that measured the response time of events published by components in the multistage workflow application using the baseline GIG middleware, *i.e.*, before integrating QED capabilities.

QED developers were also concerned about scalability since it affects the GIG middleware's data capacity while ensuring timely and reliable communication. It is also another area where QED can improve QoS relative to the baseline GIG middleware. We therefore conducted scalability tests on the baseline GIG middleware using a client/server application model that contains fewer components than the multistage workflow application in Figure 4.

Figure 5 presents the results of executing a scalability test of the GIG middleware consisting of 24 subscribers and 12 publishers publishing 2 different event types at different priorities, and deployed across 3 different hosts to distribute application workload. As shown in this figure, the baseline GIG middleware did not differentiate service based on event or workflow priority. This test therefore confirmed the QED developer's new hypothesis based on the earlier results from [3], and located another area of QoS concern that QED should address.

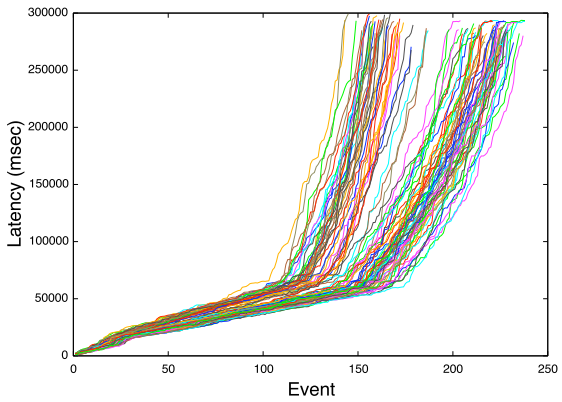

Figure 5. Evaluating scalability of GIG middleware during early stages of development.

3.2 Evaluating the CUTS's DSMLs

Applying agile techniques in DSML-based system execution tools helps reduce the effort of testing service-oriented distributed system QoS. For example, QED developers used CUTS's DSMLs to focus on modeling the behavior and workload of the multistage workflow application followed by the CUTS's model interpreters to auto-generate complete test systems. Table 1 compares the number of elements against the auto-generated source lines of codes (SLOCS) for the different test systems used by QED developers to evaluate the QoS of the GIG middleware in Sec-

tion 3.1. This table highlights how the number of modeling

Table 1. CUTS model elements vs. SLOCS of auto-generated code.

Application	Model Elements	SLOCS
Client/Server	∼20	∼530
Multistage Workflow	∼80	∼1,760

elements needed to define the test systems for the experiments conducted in Section 3.1 is substantially less than the SLOC. QED developers thus required less time and effort generating and running tests in their target environment than implementing it manually.

CUTS's agile techniques also alleviates the complexity of analyzing results for the distributed systems using UNITE (see Section 2.3). Table 2 quantifies the effort need to analyze results for both client/server and the multistage workflow application used to evaluate the GIG middleware. This table shows that the QoS unit test specifi-

Table 2. QoS unit test specification vs. analysis in UNITE.

Application	Formats:Groups	Messages
Client/Server	2:1	∼54,660
Multistage Workflow	2:1	∼236,230

cation is a lightweight process because the number of log formats and group specifications needed to process log messages that contain metrics of interest is substantially lower than the number of log messages collected during a test run that contain metrics of interest for evaluating the QoS unit test.Likewise, as system complexity grows, the number of log formats for the QoS unit test remains constant. Assuming log formats remain stable, UNITE's QoS unit test specification process is thus a one-time effort for software developers.

4 Concluding Remarks

This article showed how agile techniques realized in DSML-based system execution modeling tools, such as CUTS, help improve QoS assurance for service-oriented distributed systems throughout their lifecycle. By using DSML-based system execution modeling tools, developers need not wait until system integration time to perform critical QoS testing nor must they exert significant effort creating these tests manually. Based on our results and experience developing and applying CUTS to the QED/GIG middleware and its applications we learned the following lessons:

- **DSML-based system execution modeling tools provide practical solutions that attack the serialized-phasing development problem.** As systems grow in size and complexity, it becomes essential to alleviate the effects of serialized phasing. Our experience applying CUTS to the QED project showed that DSML-based system execution modeling tools can locate performance bottlenecks during early stages of the software lifecycle and readily adapt to different scenarios with little effort.

- **DSML-based system execution modeling tools are more cost-effective on large-scale, long-running projects.** DSML-based system execution modeling tools are most effective on large-scale projects with relatively long software lifecycles, *e.g.*, 1-2 years or more. Although learning the DSMLs does not require significant effort, the modeling process is manual and requires a dedicated set of team members to manage it. As DSML-based system execution modeling tools become more automated they will be easier to apply to projects with shorter software lifecycles and will not need dedicated team members to manage their processes.

- **DSML-based system execution modeling tools are best utilized by system integrators rather than system developers** since the overhead associated with learning system execution modeling tool DSMLs is not comparable to programmers who write source code for the target architecture. System integrators, however, often have less domain knowledge about the target architecture. The effort needed to train system integrators on the DSMLs therefore has greater payoff in the long-run, especially since models are technology-, language-, and architecture-independence and thus can be reused across different application domains.

- **DSML-based system execution modeling tools offer a cost-effective approach to validating QoS properties.** System integrators are usually responsible for validating QoS properties throughout the software lifecycle. Instead of requiring system integrators to become knowledgeable of the target architecture, DSMLs enable them to remain within their domain knowledge. Moreover, the DSMLs require less effort to realize integration tests, which helps shift system integrators effort to more important tasks, such as exploring the configuration space and its effects on QoS properties.

CUTS and its associated DSMLs are available in open-source format for download at `www.cs.iupui.edu/CUTS`.

References

[1] Department of Defense Global Information Grid Architectual Vision. www.defenselink.mil/cio-nii/docs/GIGArchVision.pdf.

[2] M. Bell. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. Wiley & Sons, Inc., 2008.

[3] J. H. Hill, H. A. Turner, J. R. Edmondson, and D. C. Schmidt. Unit Testing Non-functional Concerns of Component-based Distributed Systems. In *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation*, Denver, Colorado, Apr. 2009.

[4] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, Jan. 2003.

[5] J. Loyall, M. Carvalho, D. Schmidt, M. Gillen, A. M. III, L. Bunch, J. Edmondson, and D. Corman. QoS Enabled Dissemination of Managed Information Objects in a Publish-Subscribe-Query Information Broker. In *Defense Transformation and Net-Centric Systems*, April 2009.

[6] Pekka Abrahamsson and Juhani Warsta and Mikko T. Siponen and Jussi Ronkainen. New Directions on Agile Methods: A Comparative Analysis. In *International Conference on Software Engineering (ICSE)*, Portland, Oregon, May 2003. IEEE/ACM.

[7] M. Pezzini and Y. V. Natis. Trends in Platform Middleware: Disruption Is in Sight. `www.gartner.com/DisplayDocument?doc_cd=152076`, September 2007.

[8] Rittel, H. and Webber, M. Dilemmas in a General Theory of Planning. *Policy Sciences*, pages 155–169, 1973.

[9] D. Saff and M. D. Ernst. An Experimental Evaluation of Continuous Testing During Development. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 76–85, July 2004.

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

[11] C. Smith and L. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional, Boston, MA, USA, September 2001.