COMPUTER SCIENCE

AGILE TECHNIQUES FOR DEVELOPING AND EVALUATING LARGE-SCALE COMPONENT-BASED DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS

JAMES H. HILL

Dissertation under the direction of Professor Aniruddha Gokhale

Agile techniques are a promising approach to facilitate the development of large-scale component-based distributed real-time and embedded (DRE) systems. Conventional agile techniques help ensure functional concerns of such systems continuously throughout the software lifecycle. Ensuring quality-of-service (QoS) concerns of large-scale component-based DRE systems using conventional agile techniques, however, is hard due in large part to an observed gap between agile development and component-based software engineering (CBSE) caused by the *serialized-phasing development problem*.

This dissertation presents novel techniques in the form of algorithms, analytics, patterns, and tools to bridge the gap between agile development and CBSE for large-scale DRE systems to overcome the serialized-phasing development problem. Furthermore, this dissertation shows how our techniques can facilitate evaluation of QoS concerns continuously throughout the software lifecycle. The techniques discussed in this dissertation have been validated the context of representative large-scale component-based DRE systems from production projects in several mission-critical domains.

Approved ______________________________ Date __________________

AGILE TECHNIQUES FOR DEVELOPING AND EVALUATING LARGE-SCALE COMPONENT-BASED DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS

By

James H. Hill

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May, 2009

Nashville, Tennessee

Approved:

Professor Aniruddha Gokhale

Professor Douglas C. Schmidt

Professor Janos Sztipanovits

Professor Larry Dowdy

Professor Jeff Gray

To my family for their unconditional love and patience

and

To my friends for their never-ending support

ACKNOWLEDGMENTS

First and foremost, I would like to express great gratitude to my advisor, and superb mentor, Prof. Aniruddha Gokhale. I joined the DOC Group at Vanderbilt University under his supervision after being a Teacher's Assistant for his Operating Systems course. Throughout my tenure at Vanderbilt University, which I never will forget, his confidence and trust in me always reassured me that graduate school is where I belonged. More importantly, his never-ending support and guidance towards my research efforts, and occasional jokes (smile) during the toughest times, kept me pushing forward in the right direction.

Secondly, I would like to express great thanks to Prof. Douglas C. Schmidt. I am very grateful for the opportunity to work with such an extrodinary person. The benefits of his support on top of my advisor's support cannot be explained in words. I must say, it made me truly feel as if I was part of a family. I am also thankful for Prof. Schmidt's industry contacts, and his willingness to promote my research so that I may have real-world problems to validate my research efforts.

I would also like to acknowlege Prof. Larry Dowdy. I remember when I interviewed for graduate school at Vanderbilt University, Prof. Dowdy was the last on my list. We were schedule to meet for 30 minutes. Our meeting, however, lasted for almost 1 hour, and would have lasted longer. The joy of bouncing research ideas off one another is an experience I will never forget. I am happy to say that it was a major contributing factor in selecting Vanderbilt University for gradate school. I would also like to thank Prof. Dowdy for the fun conversations inside and outside of the classroom—especially during football season.

I would like to thank Prof. Janos Sztipanovits and Prof. Jeff Gray for serving as members of my dissertation committee. The advice and guidance they provided on my dissertation has been very helpful and will definitely inspire future work. I would also like to acknowledge Prof. Gray for his invaluable advice during the toughest part of my post-graduate career—job selection. Lastly, watching the work ethic of each person on my dissertation committee has instilled in me priceless values that I will not find anywhere else.

One of the biggest challenges of a researcher is having real-world use cases to apply their research efforts. During the summer to 2005, I had the opportunity to be a visiting researcher at Raytheon in Portsmouth, RI. During this summer, John M. Slaby was my supervisor on a DARPA ARMS project. More importantly, John was the personal investigator (PI) of the project that gave birth to CUTS, which is an artifact of my dissertation. Ever since that summer, John and has been instrumental in advocating CUTS to many stakeholders, and ensuring the CUTS vision became a reality.

I would also like to express deep gratitude to my family for their undying love and support—especially my mom and dad: Cheryl & Nathaniel H. Hill, Sr. The occasional phone call to make sure everything was going well, and the homecooked meals to ensure I was well nourished always came when I needed them the most. Dad, I hope this work makes you proud (not that you aren't already) because I just realized I am going to be an educator just like you—like father, like son.

My dissertation research was sponsored by a variety of sponsoring agencies, which include: Air Force Research Lab, BBN Technologies, DARPA, General Electric (GE) Research, Lockheed Martin ATL, and Raytheon. I would like to thank each agency for their support and willingness to sponsor my researach and provide real-world problems.

Last but not least, I would like to express great thanks to all my friends in Nashville, TN. Without you, this long journey definitely would have been a LOT harder. I especially would like to acknowledge Kenneth L. Martin, Mary E. Fleming, M.D., Kinika Young, J.D., and Gabriel L. Best, II. Your friendship and support is irreplaceable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

I.1 Emerging Trends of Large-scale DRE Systems

Large-scale distributed real-time and embedded (DRE) systems, such as air traffic control systems, mission avionics systems, shipboard computing environments, and traffic management systems, are a class of systems that are very commonplace in today's society. Large-scale DRE systems also possess serveral characteristics that contribute to their uniqueness and complexity, such as:

- Large-scale DRE systems must satisfy quality-of-service (QoS) properties, such as low latency, high reliability, fault tolerance, in addition to meeting their functional requirements [133].

- Large-scale DRE systems possess heterogeneity in both their operating environment (*e.g.*, target architecture) and techonologies (*e.g.*, programming language and middleware) [71].

- Large-scale DRE systems are traditionally developed as stove-piped, monolithic applications, which makes them very brittle, and difficult to implement, maintain, and update [107].

These characteristics as well as others not only increase large-scale DRE system complexity, but they also complicate the software lifecycle. Moreover, these characteristics and others have caused elongated software lifecycles realized at the expense of overrun project deadlines and effort [73, 116].

Due to the increasing complexity of large-scale DRE systems and their complicated software lifecycles, component-based software engineering (CBSE) [44] is becoming the *de facto* standard for developing large-scale DRE systems. CBSE is a promising approach

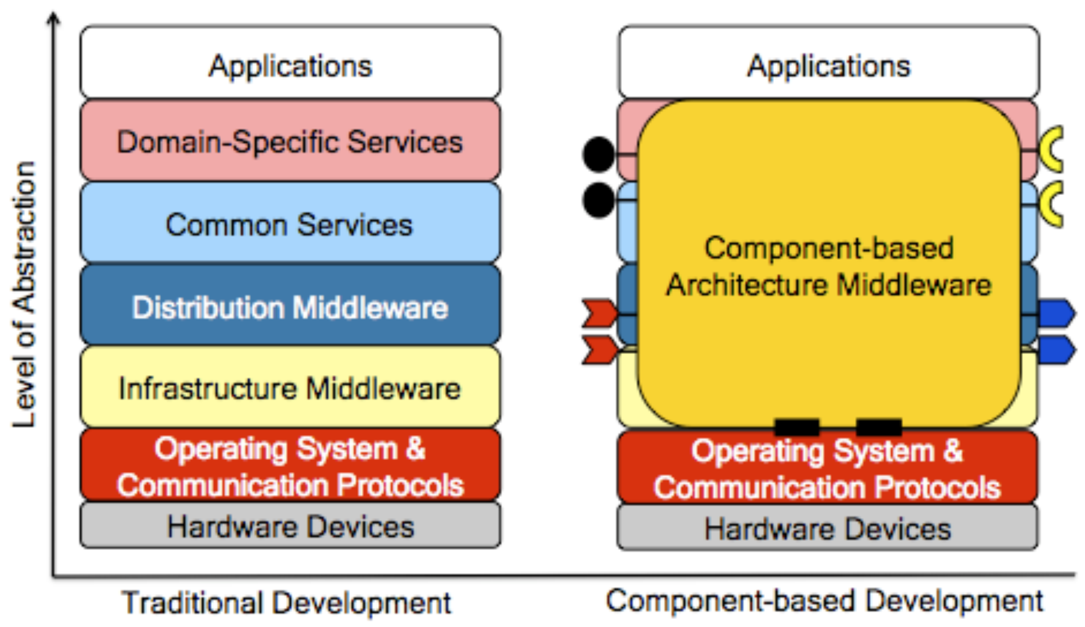

Level of Abstraction
Applications
Domain-Specific Services
Common Services
Distribution Middleware
Infrastructure Middleware
Operating System & Communication Protocols
Hardware Devices
Traditional Development
Applications
Component-based Architecture Middleware
Operating System & Communication Protocols
Hardware Devices
Component-based Development

Figure I.1: Traditional software engineering vs. componenent-based software engineering for large-scale DRE systems

for developing large-scale DRE systems because, as illustrated in Figure I.1, it raises the level of abstraction for software development. Software developers therefore focus more on the "business-logic" of the application, instead of low-level infrastructure concerns as in traditional development of such systems, such as the different layers of middleware that are implemented and managed by component-based architecture middlware. More importantly, the "business-logic" of the application is encapsulated in components, which are stand-alone entities, that can be reused across different application domains. This not only helps promote reuse of core intellectual property, but it helps expedite the software lifecycle [75], which contributes to addressing the concern of elongated software lifecyles in traditional software engineering for large-scale DRE systems.

CBSE is also enabling the realization of larger, and more complex DRE systems. Because the complexity of DRE systems is constantly increasing, it is ideal to have processes in place that ensure the system under development meets its requirements throughout the software lifecycle—especially for such systems that take many years to realize.

Agile development [97] is a promising software engineering approach for addressing such software quality assurance concerns. Agile development focuses on delivering functioning software continuously throughout the software lifecycle instead of waiting until the end of the software lifecycle to validate software functionality as in traditional software engineering. More importantly, agile development has lightweight processes that can adapt to changes in the software specification, which is commonplace for elongated software lifecycles, and ensure modifications to the software specification do not break existing functionality. Examples of conventional agile development techiqnues include, but are not limited to: continuous integration [33], Scrum [102], and test-driven development [53].

I.2 Open Issues for Agile Development of Large-scale Component-based DRE Systems

Although agile development has processes that improve functional quality assurance of large-scale component-based DRE systems, conventional agile development techniques do not address all the concerns of such systems. More specifically, large-scale component-based DRE systems are within the class of large-scale DRE systems, and large-scale DRE systems have QoS concerns that must be met in addition to their functional concerns.

As illustrated in Figure I.2, conventional agile development techniques focus primarily

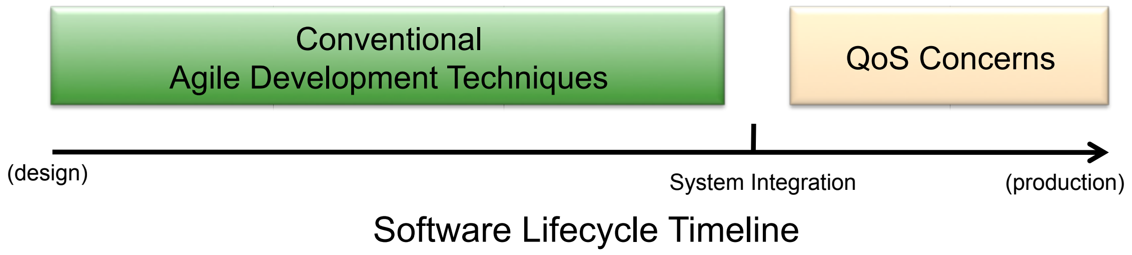


Figure I.2: Observed gap between conventional agile development techniques and QoS evaluation

on functional conerns of component-based DRE systems, and not QoS concerns. Instead, QoS concerns of large-scale component-based DRE systems are not evaluated until complete system integration time. The observed gap between agile development and evaluating QoS concerns of component-based DRE systems not only impedes seamless application of agile development for large-scale component-based DRE systems, but also makes it hard to locate and rectify problems that are negatively impacting QoS at the cost of elongated system integration phases within the software lifecycle.

I.2.1 The Effects of Serialized-phasing Development

The observed gap between agile development and evaluation of QoS concerns for large-scale component-based DRE systems stems from a development process called *serialized-phasing development*, which is a hallmark of large-scale component-based DRE system development. In serialized-phasing development, the system is developed in different layers based on its level of abstraction within the overall system, which is illustrated in Figure I.3.

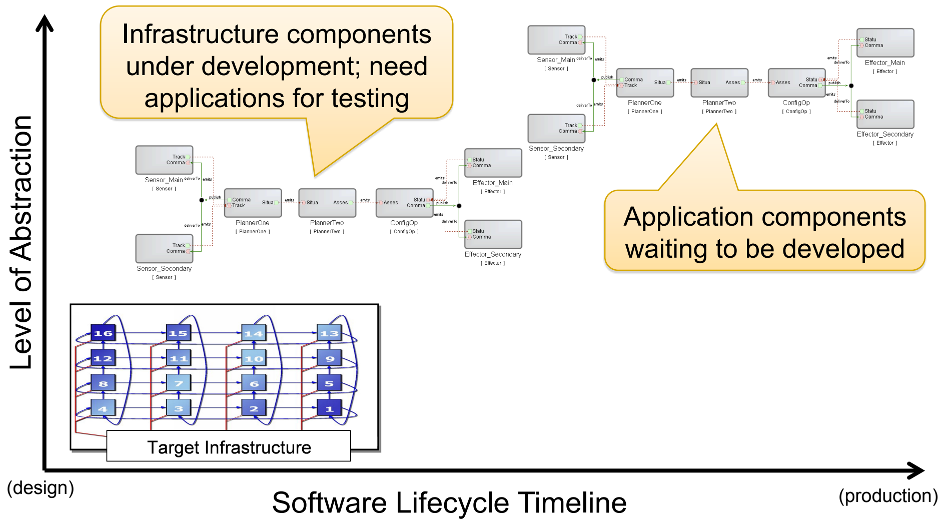

Infrastructure components under development; need applications for testing
Application components waiting to be developed
Level of Abstraction
Target Infrastructure
(design)
Software Lifecycle Timeline
(production)

Figure I.3: Serialized-phasing development in large-scale component-based DRE systems

As illustrated in Figure I.3, the target architecture, such as a representative testbed, is ready for testing and validating the system under development while the infrastruture-level components are being developed. The development of the application-level components, however, cannot begin until development of the infrastructure-level components is complete. Once development of the infrastructure-level components is complete, development of the application-level components begins.

Likewise, the infrastructure-level components are ready for testing on the target architecture, however, this process cannot begin until the development of the application-level components that utilize the infrastructure-level components is complete. After development of the application-level components is complete—which can be years into the software lifecycle—the infrastructure- and application-level components are tested together on the target architecture during a phase called *complete system integration*.

During complete system integration, software system developers and testers are able to validate the functional requirements of the system under development because existing conventional agile techniques help improve functional quality assurance continuously throughout the software lifecycle [105, 106]. Software developers and testers, however, realize their system under development fails to meet its QoS requirements. Because QoS evaluation on the target architecture does not occur until complete system integration time, it is hard to locate and rectify the QoS problems, which could be at the level of the target architecture, and/or infrastructure- or application-level components. More importantly, serialized-phasing development prevents seemless application of agile development to large-scale CBSE, and facilitation of *continuous system integration* for large-scale component-based DRE systems.

I.2.2 The Effects of Unique Deployments on Software Performance Engineering

As discussed in Section I.1, components are stand-alone entities that encapsulate core intellectual property for reuse with its application domain, and across many different application domains. Because components are stand-alone entities, this implies that within a single application, a component can be deployed to, *i.e.*, placed and executed on, any host in the target environment that can support both its functional and QoS requirements.

The nature of components also means that given a single large-scale component-based DRE system composed of many components, there will be many ways to deploy the system, which we call *unique deployments*, when realizing it within its target operating environment. Due to hardware and software contention [77, 104, 112, 114], not all unique deployments enable the system to meets its QoS requirements, such as performance.

When examining the unique deployment solution space of a given system, the number of unique deployments D can be represented by Equation I.1:

$$D = H^C \tag{I.1}$$

where H is the number of hosts in the target environment and C is the number of components in the system. [1]

Since not all unique deployments will meet performance requirments, such as end-to-end response time, software system developers and testers must rely on analytical techniques, such as software perfomance engineering (SPE), to assist in searching the unique deployment solution space and locating valid deployments that will meet their performance needs. Conventional SPE technique, such as queueing networks [77, 112], layered queuing network [129], and Petri-nets [60, 104] can suffice for analyzing performance. Such techniques, however, are parameterizable only by workload. The location of a component, or its deployment, is also a factor that can affect performance due to both hardware and software contention on the target host, which conventional SPE techniques do not take into account.

Because component location is not in input factor to conventional SPE techniques, it implies that software system developers and testers must construct individual performance

[1]Equation I.1 assumes there are no constaints on where components can be deployed and all hosts are considered unique regardless of their makeup.

models for each unique deployment of the system. As the number of host and components increases, the size of the unique deployment solution space increases. Consequently, this impedes application of conventionl SPE techniques to analyzing the performance of unique deployments. More importantly, this complexity contributes to the gap between agile development and evaluation QoS concerns of large-scale component-based DRE systems since the process is not lightweight and adaptable in favor of agile development principles [97].

I.3 Research Approach

Evaluating QoS concerns, such as performance, reliability, and security, of a system under development continuously throughout the software lifecycle is accomplished traditionally using system execution modeling (SEM) tools [112]. As illustrated in Figure I.4, while the current system is under development, SEM tools:

1. Provide techniques for validating system specification and requirements. This helps ensure that the proper system will be, and is being, developed.

2. Provide techniques for validating that the system conforms to its specification. This helps to ensure the system under development does not diverge from is original specification.

3. Provide techniques that enable software system developers and testers to conduct "what if" scenarios. This allows software system developers and testers to understand how changes at different levels of abstraction, such as specification, implementation, or operational environment, will affect the system under development.

Conventional SEM tools, however, are analytical and/or simulation model based [36, 60, 63, 113], and cannot account for all complexities of component-based DRE systems, such as the target architecture and operational environment.

Because of the shortcomings of conventional SEM tools in the context of CBSE, our approach to overcoming the gap between agile development and CBSE is to advance the state

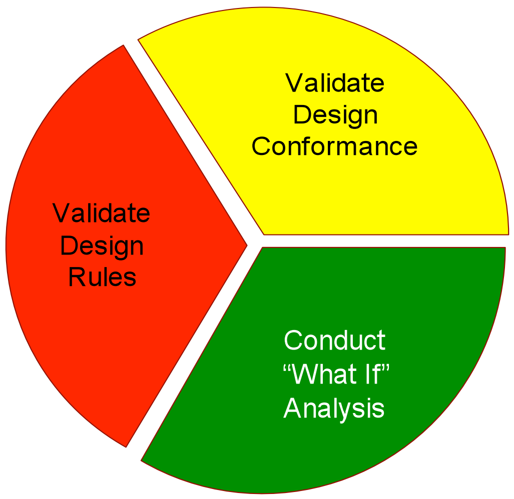


Figure I.4: Characteristics of conventional system execution modeling tools

of conventional SEM tools. More specifically, our research approach leverages domain-specific modeling languages (DSMLs) [38, 121] for enhancing SEM tools. We elected to leverage DSMLs because DSMLs capture the semantics and constraints of a target domain, such as CBSE, while providing intuitive abstractions for modeling and addressing concerns within the target domain, such as overcoming serialized-phasing development or performance evaluation of unique deployments for a large-scale component-based DRE system.

Our research approach uses DSML-based SEM tools as a bridge, *i.e.*, similar to the Bridge software design pattern [34], to fill the gap between agile development and QoS evaluation of large-scale component-based DRE systems. Our research approach using DSML-based SEM tools has resulted in algorithms, analytics, patterns, and tools for evaluating large-scale component-based DRE system QoS, such as performance, continuously throughout the software lifecyle, as opposed to waiting to complete system integration to conduct such tests.

Furthermore, our research approach has been realized in an open-source reseach artifact called *The Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS)*. As illustrated in Figure I.5, software developers and testers leverage CUTS using the following steps:

1. Model the behavior of a large-scale component-based DRE system and characterize its workload at a high-level of abstraction using DSMLs;

2. Auto-generating a complete test system for emulation that conforms to the behavior/workload model and to the target architecture;

3. Execute the auto-generated test system on its target architecture and in its target environment to generate realistic data;

4. Construct a high-level analysis model, which operates independent of data and system complexity, for processing collected data; and analyze QoS, such as performance, and determine a proper plan of action based on obtained results, such as execute more performance tests, or rectify current performance problems.

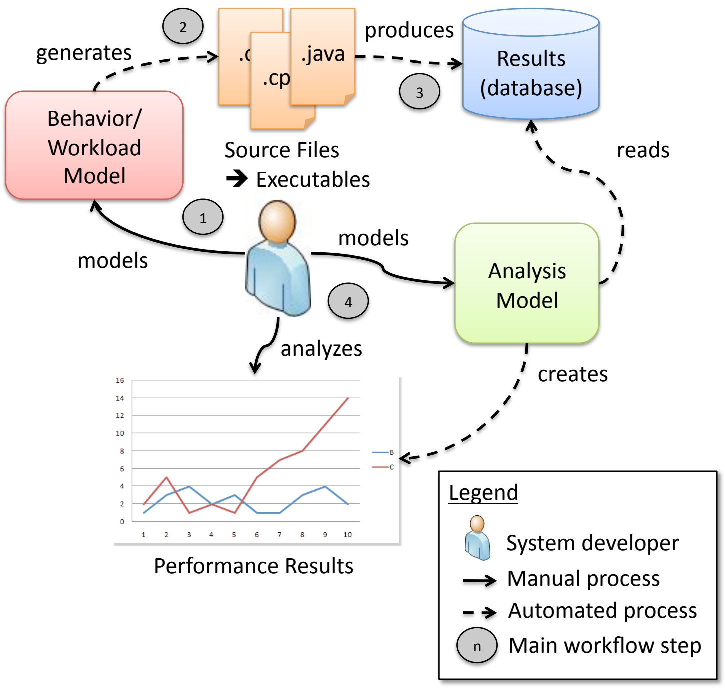


Figure I.5: Overview of the CUTS workflow

The algorithms, analytics, patterns, and tools realized in CUTS and applied in the context of representative large-scale component-based DRE systems help bridge the gap between agile development and QoS evaluation for large-scale component-based DRE systems. More importantly, CUTS facilitates evaluation of QoS concerns continuously throughout the software lifecycle, which helps increase confidence levels in quality assurance for the system under development.

I.4 Dissertation Organization

Our research on algorithms, analytics, patterns, and tools for advancing SEM tools to bridge the gap between agile development and CBSE has reduced the complexity of evaluating component-based DRE system QoS continuously throughout the software lifecycle. This dissertation is organized as follows: Chapter II discusses existing research related to the algorithms, analytics, patterns, and tools we present in later chapters; Chapter III discusses our technique for modeling behavior and workload of large-scale component-based DRE systems independent of its target architecture, platform, and technology; Chapter IV discusses our approach for using generative programming techniques to target different languages, technologies, and architectures using a single behavior and workload model; Chapter V discusses our technique for evaluating large-scale DRE system QoS independent of data and system complexity; Chapter VI evaluates different approaches for integrating DSML-based SEM tools, such as CUTS, with continuous integration environments; Chapter VIII presents our approach for using SPE analytical models to evaluate unique deployments of a component-based DRE system where both workload and deployment are input factors to the model; and Chapter IX provides concluding remarks and lessons learned from our research.

CHAPTER II

RELATED RESEARCH

This chapter discusses current research related to our research on techniques for bridging the gap between agile development techniques and continuous QoS evaluation for large-scale component-based DRE systems. The remainder of this chapter is organized as follows: Section II.1 discusses related research on behavior and workload modeling for large-scale component-based DRE systems; Section II.2 presents existing techniques for implementing reusable model interpreters for target different contexts from the same domain-specific modeling language (DSML); Section II.3 discusses existing research on unit testing and analysis for large-scale component-based DRE systems; Section II.4 presenting existing research on realizing continuous system integration testing; Section II.5 dicusses existing research for improving testing and experimentation (T&E) configurability and scalability; and Section II.6 dicusses existing research for software performance engineering of large-scale component-based DRE systems.

II.1 Behavior and Workload Modeling

This section compares our research on using DSMLs for modeling system behavior and workload to evaluate component-based systems with other related research on behavior and workload modeling for component-based distributed systems. Our research on using DSMLs for modeling system behavior and workload modeling is discussed in detail in Chapter III.

II.1.1 Behavior Modeling Languages

WinFX Workflow [15] is a process modeling language developed by Microsoft, which is a part of the Windows Workflow Foundation. Similar to our research approach, WinFX allows developers to express workflows but it is coupled with workload. WinFX also facilitates code generation, but is confined to the Microsoft .NET framework whereas our generative programming technique is technology and tool independent and can be applied to multiple middleware platforms including Microsoft .NET.

Java Workflow Tooling (JWT) [31] is a process modeling language for J2EE applications, however, it is still under development. Similar to our behavior modeling modeling approach, JWT allows developers to model the process of J2EE applications at a higher-level of abstraction using artifacts that are similar to their domain. Our research approach on behavior modeling extends JWTs efforts because, unlike JWT, our approach is not coupled to a specific programming language or technology. Moreover, our research approach has formal semantics that allow it to be used either for emulation or simulation purposes.

The Business Process Modeling Notation (BPMN) [92] is a standard developed by Business Process Management Initiative (BPMI) that allows developers to model business processes in the form of workflows. Similar to our research approach, BPMN is not platform, programming language, or technology dependent. Our research approach, however, extends BPMN by formally defining semantics based on semantic anchoring techniques so that it can leverage existing tools and techniques designed to operate on the formal language to which it is transformed.

II.1.2 Workload Modeling Languages

Executable UML (xUML) [76] and the Action Language [87] are both for defining workload that can map to the desired target architecture. Our research approach is orthogonal to both xUML and the Action Language efforts. Our research approach, however, operates at higher level of abstraction. xUML and the Action Language require developers to write abstract implementation code, which requires knowledge of programming semantics, whereas our research approach leverages pre-existing objects and methods (*i.e.*, workload generators) that are defined by the user for code generation.

II.1.3 Formal Languages

Statecharts [41] gained widespread usage when they were integrated with the STATE-MATE [42] modeling tool, and since then a variant has become part of UML (*i.e.*, UML Statecharts) [30]. Similar to our research approach, statecharts can be used to describe behavior of large complex systems. Our research approach extends Statecharts by clearly separating component behavior from workload. The generative techniques associated with variants of statecharts are targeted towards simulation and runtime verification [48, 83]. Our generative techniques can be extended to simulation and runtime verification tools [64] as well. Furthermore, our generative techniques and concepts are not tied to a specific technology or tool, whereas the technique presented in [86] et al., is bound to a specific tool.

The Abstract State Machine Language (AsmL) [40] developed at Microsoft Research is an executable specification language based on the theory of Abstract State Machines. AsmL is useful when developers need precise, non-ambiguous methods to specify a system, either software or hardware. AsmL, however, is not a graphical modeling language like our research approach. Furthermore, users of CBML operate at a high-level and do not require in-depth knowledge of the underlying formalism, whereas AsmL requires developers to have some understanding of abstract state machines and programming formalisms, which can restrict its applicability (*e.g.*, for system testers who have no knowledge of complex formalisms or programming).

II.2 Techniques for Implementing Reusable Model Interpreters

This section compares our research on addressing the limitations of the Visitor software design pattern [34], which is explained in detail in Chapter IV, with related research beyond the contexts of model interpreters. We also compare our research with other model interpretation strategies.

II.2.1 Visitor Implementation Techniques

Nguyen et al. [85] present a technique for addressing the limitations of Visitor-based implementations in the context of grammar parsers to create flexible and extensible parsers. In their technique, when a term in the grammar is visited, the parsing token uses the Abstract Factory design pattern [34] to generate the correct visitor for that token. This provides specialization on a per term basis. Our research approach is similar to theirs because we can strategize the visitation of each element type in a model (similar to a term in a grammar) and the artifacts generated from the model element. Our research approach differs from theirs since we completely decouple the generation logic from the visitation logic. When Nguyen's technique is applied to large models (*i.e.,* models with large number of elements), it suffers from the excessive memory allocation anti-pattern [112]. Our research approach does not incur this overhead.

Neff [84] and Schordan [109] present a technique for implementing a grammar parser using a variant of the Visitor pattern called the *bivisit* Visitor. This allows them to perform preprocessing/postprocessing operations before/after visiting terms in the grammar. The visitation logic however is still coupled with the generation logic. The bivisit variant of the Visitor can be used in conjunction with aspects in C++ [24] to achieve similar goals as ours. However, we are also able to separate the generation logic from the visitation logic. Moreover, our research approach can transparently customize the generation and visitation logic on a per use case basis.

II.2.2 Model Interpretation Techniques

AndroMDA [61] and oAW [94] provide tools that allow developers to implement model interpreters using template parsing and workflow engines. Developers use high-level constructs that dictate how to transform existing models into artifacts. Our research approach is similar to both AndroMDA and oAW because all three rely on generative programming techniques to implement model interpreters. We have learned, however, that reusability is less of an issue in these tools. Our research approach is tailored to DSML tools that require a Visitor-based implementation for the model interpreters.

Agrawal et al. [4] and Muller et al. [81] each present a technique that use DSMLs and model transformations to implement model interpreters. Developers use a DSML that models how to transform existing models into artifacts. Model interpreters for the transformation language then generate interpreters that perform the specified transformations. Although their technique is a level of abstraction higher, neither has explicitly addressed reusing transformations and generated interpreters within other use cases as we did with the Visitor design pattern.

II.2.3 Strategic Programming

Strategic Programming (SP) [62] is a well known generic programming technique based on programmer-definable, reusable, generic traversal abstractions that allow separation of type-specific actions from traversal specifications. Strategic programming provides a set of basic combinators, which can be composed in different ways to realize custom traversals of heterogeneous data structures. The objectives of our research approach are closely related to that of strategic programming. As such the basic primitives for handling the variabilities in the points-of-visitation could form the building blocks to define combinators. Our research approach is also tailored towards handling the variability in the points-of-generation in addition to the points-of-visitation, whereas stategic programming focuses only on the latter.

II.3 Component-based Distributed System Unit Testing and Analysis

This section compares our research on unit testing techniques for evaluating large-scale component-based DRE system QoS to related research on unit testing and component-based distributed system analysis. Our research on unit testing techniques for evaluating large-scale component-based DRE system QoS is explained in detail in Chapter V.

II.3.1 Distributed System Unit Testing

Coelho et. al [21] and Yamany et. al [139] describe techniques for unit testing multi-agent systems using so-called mock objects. Their goal for unit testing multi-agent systems is similar to UNITE, though they focus on functional concerns, whereas our research approach focuses on QoS concerns of a distributed system during the early stages of development. Moreover, Coelho et. al unit test a single multi-agent isolation, whereas our research approach focuses on unit testing systemic properties (*i.e.*, many components working together). Our research approach can also be used to unit test a component in isolation, if necessary.

Qu et. al [99] present a tool named *DisUnit* that extends JUnit [74] to enable unit testing of component-based distributed systems. Although DisUnit supports testing of distributed systems, it assumes that metrics used to evaluate a QoS concern are produced by a single component. As a result, DisUnit cannot be used to test QoS concerns of a distributed system where metrics are dispersed throughout a system execution trace, which can span many components and hosts in the system. In contrast, our research approach assumes that data need to evaluate a test can occur in any location and at any time during the system's execution.

II.3.2 Component-based Distributed Systems Analysis

Mania et. al [72] discuss a technique for developing performance models and analyzing component-based distributed systems using execution traces. The contents of traces are generated by system events, similar to log messages in our research approach. When analyzing the systems performance, however, Mania et. al rely on synchronized clocks to reconstruct system behavior. Although this technique suffices in tightly coupled environments, if clocks on different hosts drift, then the reconstructed behavior and analysis may be incorrect. Our research approach improves upon their technique by using data within the event trace that is common in both cause and effect messages, thereby removing the need for synchronized clocks and ensuring that log messages (or events in a trace) are associated correctly.

Similarly, Mos et. al [80] present a technique for monitoring Java-based components in a distributed system using proxies, which relies on timestamps in the events and implies a global unique identifier to reconstruct method invocation traces for system analysis. Our research approach improves upon their technique by using data that is the same between two log messages (or events) to reconstruct system traces given the causal relations between two log formats. Moreover, our research approach relaxes the need for a global identifier.

Parsons et al. [95] present a technique for performing end-to-end event tracing in component-based distributed systems. Their technique injects a global unique identifier at the beginning of the event's trace (*e.g.*, when a new user enters the system). This unique identifier is then propagated through the system and used to associate data for analytical purposes. Our research approach improves upon their technique by relaxing the need for a global unique identifier to associate data for analysis. Moreover, in a large- or ultra-large-scale component-based distributed system, it can be hard to ensure unique identifiers are propagated throughout components created by third parties.

II.4 Continuous System Integration Testing

This section compares our research on realizing continuous system integration testing to other related research on continuous integration environments and SEM tools. Our research on realizing continuous system integration testing is discussed in detail in Chapter VI.

II.4.1 Integrating SEM Tools with Continuous Integration Environments

Little prior work has evaluated techniques for integrating continuous integration environments with SEM tools, nor has prior work evaluated integrating SEM tools and continuous integration environments with an emphasis on improving test management. Bowyer [14] et. al discuss their experience using continuous integration environments to assess undergraduates experience using test-driven development (TDD) [53]. CiCUTS extends their effort by evaluating different techniques for integrating continuous integration environments with external processes/tools. Moreover, our research approach automatically processes collected metrics to simplify the analysis process, which Bowyer mentions as future work.

Prior work has also explored the benefits of using continuous integration environments, such as automating the build process [14], only releasing modules after they pass all automated test cases [115], and reducing integration risk by finding errors earlier in the lifecycle [46]. Our research approach also shows the benefits of using continuous integration environments to automate key aspects of the testing process. Our research approach, however, extends prior work by showing how to combine SEM tools with continuous integration environments to manage and execute performance tests that evaluate QoS. This combination allows developers and testers to focus on resolving performance issues instead of managing and executing custom test frameworks.

II.4.2 Continuous QoS Testing

The design and application of a continuous integration test suite for J2EE is discussed in [9, 128]. Although our research approach also focuses on continuous system integration testing, it combines continuous integration environments and SEM tools instead of implementing a custom environment for end-to-end system integration. Our research approach also extends the work in [9, 128] by focusing on automating the execution of a large number of tests to increase the fidelity of the QoS results, whereas [9, 128] focus on validating functional correctness.

Real-time Technology Solutions (RTTS) [12] discusses how to achieve continuous system integration testing of applications by testing at the component and system level throughout development. Our research approach is similar to RTTS in that it validates if an analyzed system satisfies its functional and performance requirements. Likewise, both

RTTS and our research approach achieve end-to-end testing by combining preexisting testing tools, such as SEM tools and continuous integration environments. Our research approach, however, extends the RTTS work by focusing on an environment for systematically executing a large numbers of tests to validate system QoS.

II.5 Techniques for Addressing Testing and Experimentation Configurability and Scalability

This section compares our work on improving testing and experimentation (T&E) configurability and scalability to existing research efforts. Our research on improving T&E configurability and scalability is explained in detail in Chapter VII.

II.5.1 Model-driven Engineering Techniques

Model-driven engineering (MDE) [108] is a common solution for improving T&E configurability and scalability. Existing MDE tools, such as GME [65], GEMS [137], and Microsoft DSL Tools [23], enable testers to construct domain-specific modeling languages (DSMLs) that capture the context and constaint of an application domain, such as T&E. Moreover, models constructed using a DSML can be tranformed by model interpreters into concrete archifacts, such as T&E configuration files. Such tools, however, facilate generation of single instance configuration files only.

Our research approach extends existing MDE techniques by replacing single instance configuration files with template files, and delay realization single instance configuration files using their template patterns. Furthermore, our research approach synergizes with existing MDE techniques because it is possible to generate the configuration templates from constructed models. This, therefore, reduces the number of models that tester must create to evaluate a component-based DRE system under different configurations and operating scenarios (or environments).

II.5.2 Programmatic Techniques

Template libraries, such as Google Templates [37] and CodeSmith [125], enable developers to programmtically construct template engines for generating files, such as T&E configuration files. End-users, such as testers, then define dictionaries to derive concrete files via the Template Configuration pattern. In a similar fashion, our research approach enables derivation of single instance configuration files using the Variable Configuration Pattern. Our research approach extends existing template engines by enabling batch processing of configuration templates. We are aware that such support can be added to existing template engines by handling multiple configurations sequentially.

CodeSmith also has the notion of what we would consider dynamic variables in its template engine. Unlike our research approach, CodeSmith evaluates its dynamic variables outside of its target environment, such as the testbed for T&E. Our research approach extends CodeSmith's effort by allowing evaluating dynamic variables based on its target operating environment. Consequently, this improves both the configurability and flexibility of our research approach's template engine above and beyond CodeSmith's capabilities.

II.6 Software Peformance Engineering for Component-based DRE Systems

This section compares our research on software performance engineering (SPE) techniques to other related research on evaluating large-scale component-based DRE system QoS. Our research on improving SPE techniques to evaluate large-scale component-based DRE systems is discussed in detail in Chapter VIII.

II.6.1 Compositional Performance Model

Hissam et. al [45] demonstrated the feasibility of using a compositional model, *i.e.*, performance predictions based on how components are assembled and taking into account their structure, to predict the performance of component-based systems. Their technique was applied to the COMTEK component architecture, which was developed the US Environmental Protection Ageny (EPA) Department of Water Quality. Their approach augmented the modeling tools for the technology to support prediction of latencies using quantitative analytical models. Based on their results, Hissam were able to predict the latency of assemblies within 10% error.

Bondarev et. al [13] also demonstrated the feasibility of using a compositional model to predict resource usage of component-based real-time systems for predicting end-to-end response times. Their approach extends a existing scenario-based approach by allowing developers to model the behavior of the application and underlying component technology. The results of their analysis determine if the constructed design will meet end-to-end performance requirements so developers can move onto the implementation phase of the system.

II.6.2 Queuing Network Model

Liu et. al [67] demonstrated that it is possible to construct traditional queuing network models for service- oriented architectures, such as CORBA Component Model [93], Enterprise Java Beans [120], and Microsoft.NET [79]. Their approach focuses on predicting the performance of systems that communicate via their exposed services, which is analagous to a remote method invocation. Liu first benchmark the system, which includes benchmarking the component's underlying architecture. Once this process is completed, they are able to construct a quantitative model of the system that can be parameterized to predict the performance of the system. Their approach predicts the performance of the system within 11% error of the actual performance.

II.6.3 Event-based Performance Model.

Liu et. al [68] extended their previous efforts to constructing queuing network models for component-based systems that communicate asynchronously using events. Using their technique for constructing a qeueing network model, Liu are able to predict performance within 10% error with a maximum error of 15%, which is claimed to occur infrequently. Their approach assumes that all components in the server are collocated, which implies work between components in the system is not truly distributed. Moreover, if the deployment changes, their approach requires benchmarking the new deployment before parameterizing the model to predict its performance.

CHAPTER III

DOMAIN-SPECIFIC MODELING LANGUAGES FOR OVERCOMING SERIALIZED-PHASING DEVELOPMENT

Model-driven engineering (MDE) [108] techniques, such as domain-specific modeling languages (DSMLs) [38], are increasingly being used to address many of the development and operational lifecycle complexities of large-scale component-based DRE systems. Although there have been many advances in MDE for large-scale component-based systems, MDE techniques to date have focused primarily on (a) structural issues of system development, such as component assembly, packaging, configuration and deployment [35, 94, 135], and (b) functional and behavioral issues, such as model checking for functional correctness (*e.g.*, Bogor [103] and Cadena [43]) or runtime validation of performance (*i.e.*, running empirical benchmarks at integration time to validate performance).

Although DSMLs raise the level of abstraction of large-scale component-based DRE systems and address many of their complexities, there remains a major gap in evaluating system quality of service (QoS), *e.g.*, performance and reliability, continuously throughout the software lifecycle, which would enable design flaws to be rectified earlier in the development lifecycle. This impediment is due primarily to the serialized-phasing development problem within the software lifecycle. As explained in Chapter I, during serialized-phasing development, the system is developed in layers (*e.g.*, first the components at the infrastructure layer(s) and then the application layer(s)). Throughout the development of each layer, the business-logic encapsulated within individual components is thoroughly tested for functional correctness (*i.e.*, whether it performs the expected operations).

Due to the composite nature of large-scale component-based DRE systems, complete system QoS validation can proceed only when all the system components are available and deployed in the runtime infrastructure [66]. Consequently, waiting too late in the development lifecycle (*e.g.*, integration time when all components are available) to resolve any performance problems can be too costly to resolve. System developers and testers, therefore, need improved techniques and methodologies to help address QoS validation not only at complete system integration and production time, but continuously throughout the software lifecycle before performance problems become too hard to locate and resolve.

Solution approach: Validating QoS via emulation techniques. A promising solution to address the challenge of evaluating QoS at all stages of development entails accurately emulating system components for QoS validation while the "real" components are being developed. This chapter describes our novel MDE-based solution to address the challenges of serialized phasing development and QoS validation continuously throughout the software lifecycle.

At the core of our solution are two DSMLs named the Component Behavior Modeling Language (CBML) and the Workload Modeling Language (WML), which allow developers to define the behavior and workload, respectively, of large-scale component-based DRE systems at a higher-level of abstraction from that provided by third generation programming languages, such as C++, C# and Java. The behavior in CBML is captured using formalisms of Timed I/O Automata (TIOA) [70] and can be parametrized with executable operations (*i.e.*, workload) from WML. Our DSMLs, however, do not require system developers and testers to possess *a priori* knowledge of TIOA. Lastly, we illustrate how we are able to preserve the sematics of TIOA at the modeling level using a technique called *semantic anchoring* [18, 19, 82], which uses well-defined transformations to map a DSML to an existing formal language (*e.g.*, Timed I/O Automata [70] and timed-automata [7]) to validate and formally define the semantics of the DSML.

Chapter organization. The remainder of this chapter is organized as follows: Section III.1 presents modeling challenges for overcoming serialized-phasing development;

Section III.2 describes the structure and functionality of our DSML for emulating component behavior and workload; Section III.3 explains how we integrate our DSMLs with existing structural DSMLs to associate behavior and workload models with structural models; Section III.4 explains how we use semantic anchoring to preserve the semantics of TIOA at the modeling level; Section III.5 explains how we facilitate code generation for QoS validation using emulation techniques; and Section III.6 summarizes the contributions of this chapter.

III.1 Case Study: A Distributed Stockbroker Application

This section describes the challenges in developing a solution that addresses the need for early QoS evaluation of large-scale component-based DRE systems affected by serialized-phasing development. We use a representative example taken from the financial domain [122] as a motivating example to illustrate the serialized phasing problem and how our research artifacts described in this paper enable us to provide early QoS validation. Our case study is called the Distributed Stockbroker Application (DSA), which is an online web application for viewing stock information.

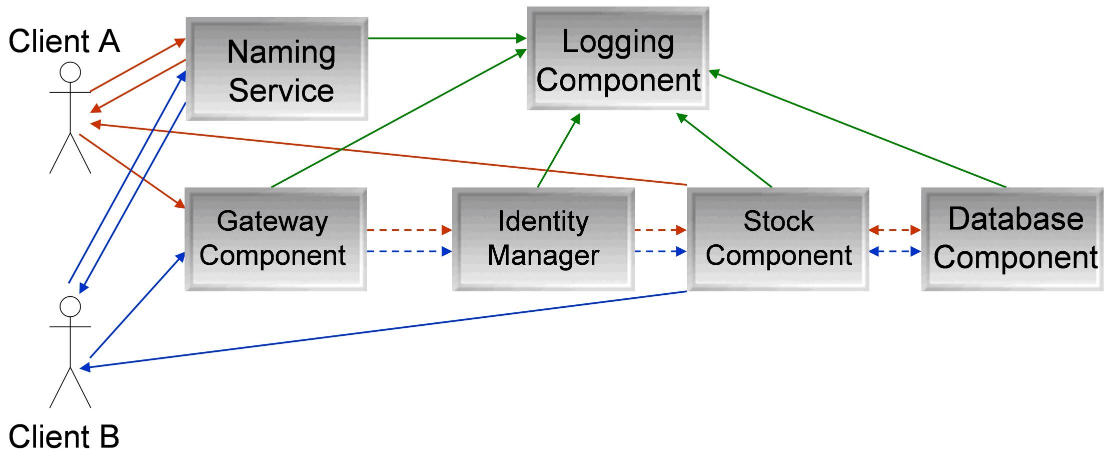


Figure III.1: High-level structural composition of the Distributed Stockbroker Application

Figure III.1 shows a high-level representation of the DSA and its communication flows between components. The DSA is composed of six different components. The *Naming Service* component allows client applications to locate the *Gateway Component* for the application. The location (*i.e.*, the binding IP address and port number) of the naming service component is therefore persistent. The Gateway Component serves as the entrance to the stock application, which all clients must pass through. The Gateway Component accepts the username and password of the user and sends it to the *Identity Manager* component. The Identity Manager component is responsible for verifying the username and password, and initializing the correct QoS policies based on user type. Once the access is granted to the client, it is given direct access to a *Stock Component*. The Stock Component is created on-demand and initialized with the correct QoS specified by the Identity Manager. The Stock Component interacts with a MySQL database that contains the stock information. Lastly, all components in the system—both application and infrastructure—log their activities to a *Logging Component*.

The DSA has two user classes: *Basic* and *Gold*. Gold users are persons who use the service frequently, whereas Basic users use the service infrequently. Table III.1 provides the projected usage pattern and desired response times (*i.e.*, QoS) of each user for the DSA. Due to the serialized-phasing development process, the underlying infrastructure of the DSA (*i.e.*, all the components illustrated in Figure III.1) may complete their development at different times in the software lifecycle. Evaluating system design decisions on the target architecture to understand and evaluate system QoS, therefore, has to wait until all the "real" components are available.

Table III.1: Predicted usage pattern of the Distributed Stock Application based on user type

Type	Percentage	Response Time (msec)
Basic (Client A)	65%	300
Gold (Client B)	35%	150

The application components of DSA are implemented as Lightweight CORBA Component Model (CCM) [91] components. The target architecture comprises three hosts for deploying all its components. Lastly, the software platform version is Fedora Core 4 using ACE+TAO+CIAO 5.1 middleware platform available at `www.dre.vanderbilt.edu`.

To achieve the vision of early QoS validation in the presence of serialized phasing, such as in the case of the DSA case study, the proposed solution must address the following challenges:

- **Challenge 1: Capture business logic** – The components must resemble their counterparts in both supported interfaces and behavior. For emulation, the target environment should allow seamless replacement of faux components with real components as they become available. For simulation, however, seamless replacement is not applicable. The configuration files for simulation must define elements (*e.g.*, inputs, outputs, and transitions) that resemble their real counterpart to preserve similarity and contextual representation.

  In the context of the DSA, emulated components should be used to evaluate QoS at early stages of development, and as the "real" components are available they should replace the emulated components to achieve more accurate QoS metrics. Likewise, the simulated components should be used to verify properties such as functional correctness and reachability.

- **Challenge 2: Realistic mapping of behavior** – The behavior specification should operate at a high-level of abstraction (*i.e.*, at the application level) and map to realistic operations (*e.g.*, memory allocations and deallocations, file operations, or database transactions).

  For example, in the DSA, the high-level database behavior should "realistically" query a database for stock information when using emulation. In the context of simulation, the behavior should map to well-defined elements of the underlying formal language that represent querying a database.

- **Challenge 3: Technology independence** – The behavior specification should not be tightly coupled to a programming language, middleware platform, hardware technology, or MDE tool.

  In the context of the DSA, if we wanted to evaluate the system on CCM or Microsoft .NET [78], or use multiple modeling tools [65, 136], then we should be able to reuse the same concepts and models. Likewise, if we wanted to simulate the DSA under different tools such as Tempo `(www.veromodo.com)` or UPPAAL `(www.uppaal.com)`, we should be able to reuse the same models.

The remainder of this chapter describes our solution to resolve these challenges.

III.2 Modeling Component Behavior & Workload for QoS Validation

Addressing the challenges of continuous system integration and QoS evaluation in the face of serialized-phasing development requires mechanisms to mimic application component behavior. This section describes our research approach on two DSMLs named the Component Behavior Modeling Language (CBML) and the Workload Modeling Language (WML). CBML is a DSML for capturing the behavior of a component, and WML is a DSML for parameterizing the behavior with "realistic" application-level operations. The remainder of this section discusses both languages in detail explaining how these help resolve the challenges presented in Section III.1.

III.2.1 The Component Behavioral Modeling Language

Any mechanism that mimics component behavior must incorporate the design principles and semantics of component architectures. In such architectures, systems are composed of components that react to method invocations and events received on their input ports. This "reaction" causes a sequence of activities that can be defined by a series of states and transitions. Although the range of activities performed in the course of a component's execution can vary broadly, they can be divided into two distinct operational classes: *internal* and *communication*.

Internal operations are those not observable from outside a component (*e.g.*, memory allocations/deallocations and database transactions executed by the database component in the DSA case study). Communication operations are representative of sending/receiving an event to/from another component (*e.g.*, input and output events transmitted between each of the components in the DSA case study).

When trying to emulate a component's behavior (*i.e.*, addressing Challenge 1 in Section III.1), it is desirable to capture it as close as possible to its real counterpart using combinations of internal and communication operations. It is also desirable to represent the behavior based on a formal mathematical foundation because it will (1) facilitate transformation of existing models between different formal behavioral languages (*e.g.*, timed-automata, StateCharts [41] and Petri Nets [98]), and (2) assist in proving any formal properties of the system (*e.g.*, correctness and stability). Likewise, it will also facilitate reverse transformations (*i.e.*, from models in other languages to models of this language). We believe that lack of formal semantics can limit the capabilities and scope of such behavior modeling languages. At the same time, it should not be dependent on any programming language or software/hardware platform, and be as general purpose as possible.

Based on the desired functionality for modeling component behavior, we have researched and designed CBML. CBML is a DSML based on the mathematical formalism of Input/Output (I/O) automata [70] [1]. We chose I/O automata as its foundation because, analogous to component behavior, I/O automata is ideal for asynchronous and reactive systems. We developed CBML in the Generic Modeling Environment (GME) [65], which is a metamodeling environment that allows the creation of DSMLs and its models. CBML, however, is not coupled to GME, and can be ported to any MDE tool that supports metamodel specification (*e.g.*, Eclipse Modeling Framework (EMF) [16], Generic Eclipse Modeling System (GEMS) [136], or Microsoft DSML tools [39]). Developers use CBML to capture component behavior at a high-level of abstraction and use model interpreters to generate configuration and source files for backend emulation tools, which is explained in detail in Chapter IV.

III.2.1.1 Structure of CBML

As explained in Section III.2.1, we developed CBML based on the mathematical formalism called I/O automata [70]. We, therefore, defined CBML so that it has the necessary subset of elements from I/O automata that will preserve its formal semantics. Users of CBML do not need prior knowledge of I/O automata to use CBML. Keeping that in mind, we formally define the behavioral model $BM = (V, S, \Theta, I, O, A, E, T)$ of a component in CBML as:

- a set V of *internal variables*,
- a set $S \subseteq val(V)$ of *states* where $val(V)$ is the value of the internal variables at any given point in time, or the current state of the component,
- a nonempty set $\Theta \subseteq S$ of *start states*,
- a set I of *input actions*, which are events received from an external source, *e.g.*, a connected component,

[1]The details of I/O automata are beyond the scope of this dissertation.

- a set O of *output actions*, which are events sent to an external destination, *e.g.*, a connected component, and
- a set A of *actions*, which are events (or actions) visible only to the component hosting the behavior, *i.e.*, internal operations.

Figure III.2 highlights each of these elements in *BM* as their representative artifacts in CBML.

Input Action
State
Action
Output Action
Variable

Figure III.2: Primary elements for constructing behavioral models in CBML

In order to construct valid behavioral models in CBML, developers must specify a sequential flow between different actions $\Omega = (I \cup O \cup A)$ and states S. I/O automata partially supports this requirment via its set of effects E, which determines how to move from a given action to a new state and is defined in Equation III.1 as:

$$\Gamma(a) \rightarrow s, \tag{III.1}$$

such that $\Gamma \in E$ and $a \in Omega$.

On the other hand, I/O automata, has no notion of action sequencing since actions are always enabled and can occur at any time. We, therefore, extended I/O automata at the modeling level, *i.e.*, within CBML, to support a concept called *transitions*. Transitions determines how to move from a given state to another action, and is defined by Equation III.2:

$$\Delta(s) \rightarrow \alpha, \tag{III.2}$$

such that $\Delta \in T$ and $\alpha \in (A \cup O)$.

Realizing behavior models in CBML. Figure III.3 shows the complete realization of *BM* using the respective CBML artifacts illustrated in Figure III.2, Equation (III.2) and Equation (III.1) in the context the DSA database component. In CBML, all behavior specifications begin with an Input Action element. Each Input Action in the behavior model is connected to an initial State element. The remainder of the behavioral specification is defined by a sequence of Action to State transitions. For example, the behavioral model for the database component in Figure III.3 illustrates that an input action causes a query for stock information.

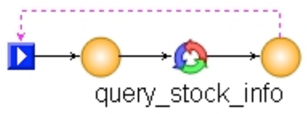


Figure III.3: Example CBML behavioral model in GME

To specify the end of a behavior sequence in the modeling realm, a *Finish* connection (*i.e.*, the dashed line) is used to connect the final *State* to the starting *Input Action*. We require this connection because we allow sharing of behavioral sequences to simplify modeling (illustrated in Figure III.4). For example, the DSA has two type of users who have the same behavior. It is possible to model each person's input to the database component (or any component) separately but share the same behavior as illustrated in Figure III.4.[2] The explicit finish connections therefore help resolve ambiguity when determining where each user type's behavior terminates.

During the interpretation process of CBML, we treat shared behavioral sequences as separate sequential flows to preserve the validity of Equation (III.2) and Equation (III.1).

[2]Shared behavior is a modeling optimization we allow to help reduce the size of constructed models because automata-based models are affected by state-space explosion as they grow in the number of elements.

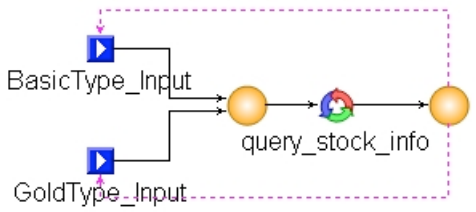


Figure III.4: Example of sharing behavior in CBML

Figure III.5 illustrates how Figure III.4 is handled during the interpretation process to generate either emulation or simulation files. As shown in Figure III.5, there are now 4 different states and 2 different actions, thus preserving the validity of Equation (III.2) and Equation (III.1).

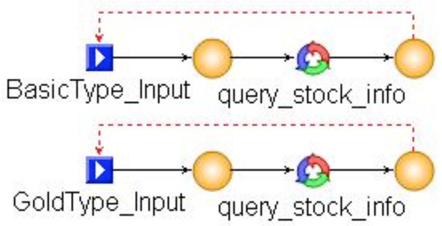


Figure III.5: View of shared behavior in CBML from the interpretation perspective

Specifying output actions in CBML. CBML defines behavior as an input action that causes a series of "internal" operations and results in a set of output actions, if any. Based on the definitions of a transition Δ from Equation (III.2), it is clear that output actions O have the same modeling semantics as actions A.

Figure III.6 illustrates an example behavioral model with output actions, which are represented by the three rightmost squares labeled `basic_response`, `gold_response` and `log_status` for the database component in the DSA. After the component completes its query to the database for stock information, it outputs information back to the requester, and outputs a status message to the external logging component.

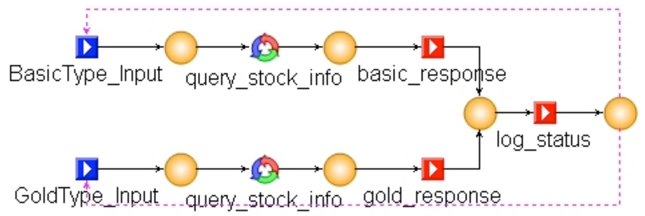


Figure III.6: Example CBML behavioral model with output actions

Preconditions, postconditions, and variables in CBML. CBML allows users to define variables V in behavioral models to preserve information that represents the current state of the component, $val(V)$. Preconditions, which are associated with transitions Δ, operate on the variables to enable and/or disable the execution of individual transitions. Likewise, postconditions, which are associated with effects Γ, modify the values of variables to change the current state of the component, or system. Formally, preconditions and postconditions are defined as follows:

- For preconditions:

$$\Delta(s) \leftrightarrow pre(val(V)) \tag{III.3}$$

  where $pre(X)$ determines if the current value of X is `true`.

- Upon execution of effect Γ associated with action a

$$post(a) \rightarrow val(V)', \tag{III.4}$$

  where $a \in \Omega$ and $val(V)'$ is the new state of the system such that $S \subseteq val(V)'$.

As illustrated in Figure III.7 in the context of CBML, a variable is represented by the element with the star image. Users use variables in their behavioral model by referencing them in the preconditions and postconditions of the transition (*i.e.*, connection from a state to an action), and effect (*i.e.*, connection from an action to a state) connections, respectively.

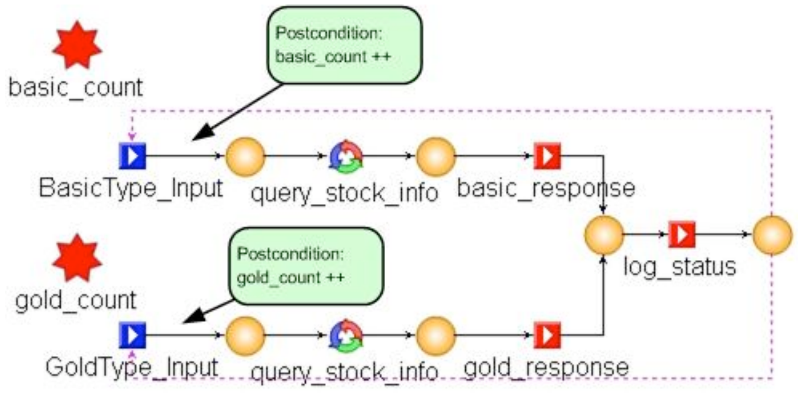


Figure III.7: Example CBML behavioral model with variables

This allows developers to create more "realistic" behavioral models, such as counting the number of users of each type executing queries on the database and/or guarding a workload until the system reaches a certain state.

Domain-specific extensions in CBML. Some input events that are critical in the domain of component-based systems (*e.g.*, lifecycle events such as *activation* and *passivation* or monitoring notification events such as degradation of QoS) are not first class entities in I/O automata. I/O automata does not distinguish between these kinds of events because it is a general-purpose language that is not tied to any particular domain (*e.g.*, component-based systems). We therefore extended I/O automata (without affecting its formal semantics) in CBML to capture this aspect of component behavior more expressively as discussed below and illustrated in Figure III.8:

- **Environment events**, $E \subseteq I$, represent input actions to a component that are triggered by the hosting system rather than another component (*e.g.*, lifecycle events from the hosting container or fault-tolerance notifications to serialize the state of a component).

- **Periodic events**, $P \subseteq I$, represent input actions from the hosting environment that occur periodically (*e.g.*, setting/receiving a timeout event to periodically transmit status updates). We also allow a distribution class to be associated with periodic events, such as constant or exponential distribution.

- **Application task**, $AT \subseteq I$, represents a one-time occuring input action into a component on a separate thread of control.

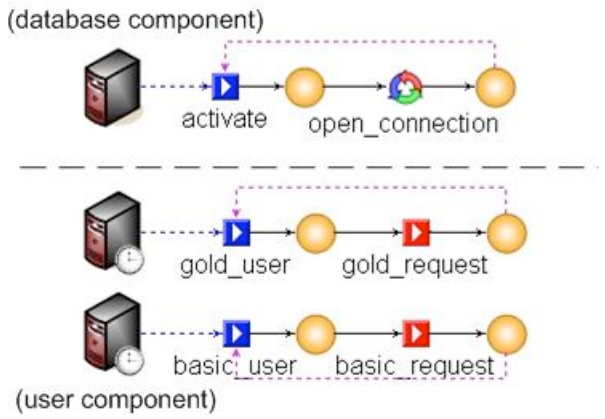


Figure III.8: CBML's domain-specific extensions to I/O automata

In the context of the DSA, when the database component is activated it creates an initial connection to the database (illustrated in upper half of Figure III.8). Likewise, we can use periodic events (illustrated in the lower half of Figure III.8) to model the behavior of each user type by associating each one with correct probability (*e.g.*, 0.35 and 0.65 for Gold and Basic type, respectively) and sequencing it with an output event within a "user" component (also illustrated in Figure III.8).

Usability extensions in CBML. One of the main goals of defining behavior at a high-level of abstraction is simplicity and ease of use. If the size of the behavioral model is "huge" and CBML adheres strictly to its current representation of I/O automata, its ease of use is compromised because one of the major drawbacks of many automata languages is scalability [41]. To address this issue we defined the following usability extensions, which do not violate the definition of BM in CBML:

- **Composite Action**, $CA \in A$, is a modeling element that allows developers to create reusable behavior workflows that can help reduce the amount of clutter in the model. A composite action has the same definition as BM. We, however, define a constraint that requires composite actions to contain only a single input action, *i.e.*, $|I| = 1$. This is necessary because composite actions encapsulate a single, reusable behavior workflow and not multiple behavior workflows.

- **Log Action** is an attribute of an *Action* element that determines if the action should be logged. The semantics of "logged" are dependent on how the model is interpreted. For example, a modeler might choose to log "network send" actions and not "memory allocation" actions.

To address the usability concerns in the modeling aspect, we also leveraged GME add-ons that assist users in creating models rapidly by auto-generating required elements (*e.g.*, states) and connections depending upon the context. Although this feature is GME-specific, most MDE tools provide support for implementing features that help improve the user experience [127].

III.2.1.2 Supporting Timing Semantics in CBML

I/O automata is ideal for modeling asynchronous, reactive systems, such as large-scale component-based DRE systems. When trying to evaluate QoS, however, I/O automata lacks several aspects, such as timing, that would allow developers to verify QoS properties about components, and the system (*e.g.*, end-to-end deadlines, expected execution time, etc.).

To address this limitation, Timed Input/Output Automata (TIOA) [57] was defined as an extension I/O automata to support timing aspects. TIOA has the same formal semantics as I/O automata, but it is extended to support both discrete and continuous variables. The continuous variables (*e.g.*, a clock or temperature) define how the state of the system changes with respect to time.

Because CBML was originally based on the semantics of I/O automata, it also lacked the same properties that would allow developers to validate QoS properties from a simulation standpoint. We, therefore, extended CBML to support the notion of timing to be consistent with TIOA. In CBML, timing is defined by the following equation:

$$clock' = clock + time(a) \tag{III.5}$$

where $clock$ is the current timing variable for the component, $time(a)$ is the execution duration of $a \in A$, and $clock'$ is the new clock time after completing a. In CBML, we only associate timing with internal actions A because we, currently, make the assumption that all input I and output O actions are instantaneous.

III.2.2 The Workload Modeling Language

CBML described in Section III.2.1 gives developers the ability to model behavior via generic actions and properties. For analysis techniques, such as simulation, CBML is enough to capture the behavior of the component (*e.g.*, its actions, states, and respective transitions), which can be interpreted to define configuration files for simulation tools. For emulation purposes, however, these actions do not exemplify the "business logic" of components because they do not capture the workload of reusable objects within a component (*e.g.*, objects and their methods). Moreover, when defining the workload of components using CBML, it is hard to specify realistic workloads that map to executable operations for an emulated component. To address this challenge (*i.e.*, Challenge 2 in Section III.1) we developed the Workload Modeling Language (WML).

WML is a middleware and hardware platform-independent and programming language-independent DSML that allows developers to define workload generators that contains actions to represent realistic operations (*e.g.*, memory allocations/deallocations and database transactions) at a high-level of abstraction. Model interpreters associated with WML parse the constructed models and use generative programming techniques to map the abstract representation to executable operations in the target programming language and platform (see Section III.3).

We implemented WML in GME, but similar to CBML, it can be ported to any MDE tool that supports metamodel specification. The remainder of this section discusses WML in detail.

III.2.2.1 Structure of WML

WML is a DSML that allows developers to create workload generators (called *workers*) with executable actions for emulation. Figure III.9 illustrates the compositional overview of WML. We designed WML using a hierarchical structure that resembles common object-oriented programming techniques to be consistent with conventional component technologies.

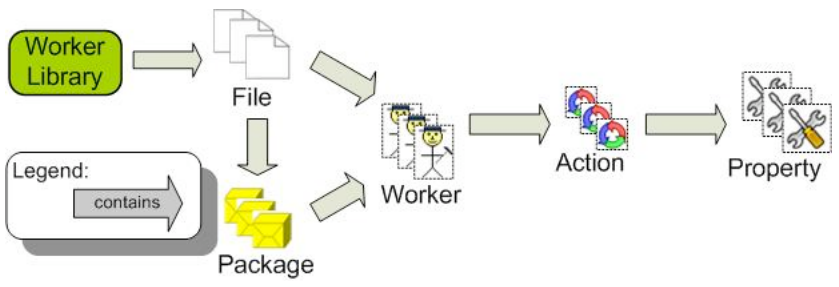


Figure III.9: High-level compositional overview of WML

The outermost containment elements in WML are *library* elements. Library elements represent reusable containers (*e.g.*, modules) for grouping common *workers*. The library elements are composed of multiple *files*, which represent a concrete location on disk that defines its contained workers (recall that workers are workload generators). File elements can contain *packaging* elements that act as a scoping mechanism so that *workers* can have the same name and appear in the same file (similar to C++ namespaces or Java packages).

Workers contain executable actions that represent its "business logic" operations (or workloads). Lastly, actions can contain optional *properties* that define configurable parameters (*e.g.*, input arguments) for the action executed by the parent.

III.2.2.2 Parameterizing CBML with executable operations

When WML is integrated with CBML, it enables developers to model the component behavior using executable operations. From a modeling perspective, *workers* in WML have the same modeling semantics as *variables* in CBML, and worker *actions* in WML have the same modeling semantics as *actions* in CBML. This design feature allows us to integrate WML with CBML.

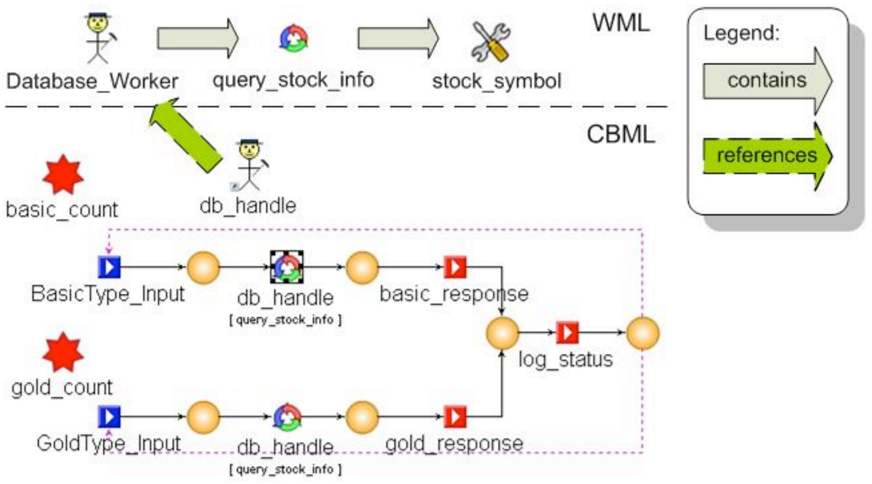


Figure III.10: Example CBML model parameterized with WML actions

Figure III.10 illustrates the behavior model of the database component from Figure III.7 in Section III.2.1.1 that has been parameterized with WML actions. The top portion of the image illustrates the WML composition for a database worker. In the bottom portion of the image, the actor (*i.e.*, `db_handle`) is a *variable* that references the database worker. The *action* is a modeling instance of the worker action in the top portion of the image whose name must match the name of its parent worker variable. We made this design requirement because it (1) helps resolve ambiguity when determining what action belongs to what parent since it is possible to include the same worker variable type multiple times in a behavior model, and (2) reduces modeling clutter as opposed to explicitly creating a directed connection between a parent and its action.

III.3 Integrating Behavioral and Structural DSMLs

In Section III.2, we described a behavioral DSML named CBML and illustrated how it allows us to capture the behavior (Challenge 1 of Section III.1) and map the behavior to realistic operations (Challenge 2 of Section III.1). Although CBML allows us to capture the behavior of a component, the models are insufficient to generate simulation code directly without knowing the structural composition of the system and its components for QoS validation since the latter determines the end-to-end workflows.

We, therefore, integrated CBML with the Platform Independent Component Model Language (PICML) [10] because PICML captures the structural aspects of large-scale component-based DRE systems, such as deployment, interface definitions, and packaging. Moreover, since both PICML and CBML provide platform and programming language independent modeling capabilities, their integration and model interpretations provide a technology independent approach to continuous QoS evaluation (Challenge 3 in Section III.1). Although we chose PICML as the structural DSML to integrate CBML, our integration concepts can be applied to any structural DSML—provided the structural DSML clearly differentiates between input and output ports of a component.

When we examine structural DSMLs, such as PICML, it facilitates modeling different ports of a component (*e.g.*, facet/receptacles and event sources/sinks). The facets/event sinks represent inputs to a component, while receptacles/event sources represent outputs from a component. We, therefore, can formally define the structural aspect of a basic component $C = (M, N)$ as:

- a set M of input ports for receiving events from external sources, *e.g.*, connected components, and

- a set N of output ports for sending events to external destinations, *e.g.*, connected components.

Structural DSMLs, however, capture structural input/output (I/O) elements without any correlating behavior (*i.e.*, there is no clean representation to associate the I/O elements of structural models with the I/O actions in behavioral models). We, therefore, extended the structural definition of a component $C = (M, N, \Phi, \Psi)$ to define a set of functions that enable developers to connect the I/O elements in the structural model with corresponding I/O elements in the behavioral model BM (see Section III.2.1.1) based on the following equations:

- Let $m \in M$, $i \in I$ and $\phi \in \Phi$, then

$$\phi(m) \rightarrow i. \tag{III.6}$$

- Let $n \in N$, $o \in O$ and $\psi \in \Psi$, then

$$\psi(o) \rightarrow n. \tag{III.7}$$

Figure III.11 illustrates how structural DSMLs (*e.g.*, PICML) that define components that have I/O ports and behavioral DSMLs (*e.g.*, CBML) that have I/O actions can be integrated by having the structural DSML "contain" the behavioral DSML and applying Equation (III.6) to the structural DSML and Equation (III.7) to the behavioral DSML.

In the modeling realm, we require a component to contain the behavior. Additionally, we define a modeling connection between the input port and input action to implement Equation (III.6), but require that the name of the output action match the name of the corresponding output port to implement Equation (III.7). We made this design decision

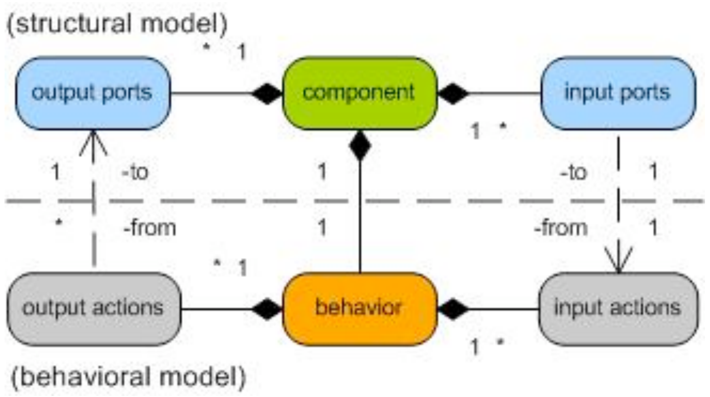


Figure III.11: Conceptual model of integrating behavioral and structural DSMLs

because explicitly defining a connection between an output action and port will clutter the model since there is a many-to-one mapping between an output action and an output port.

To further illustrate this concept, Figure III.12 shows the realization of integrating a behavioral and structural DSML. The outer rectangle of Figure III.12 illustrates the PICML model of the database component. The inner rectangle highlights the same database component with CBML from Figure III.7 integrated into PICML, thus allowing us to model the same behavior exemplified in Section III.2 with its respective structure (*e.g.*, interface and attributes).

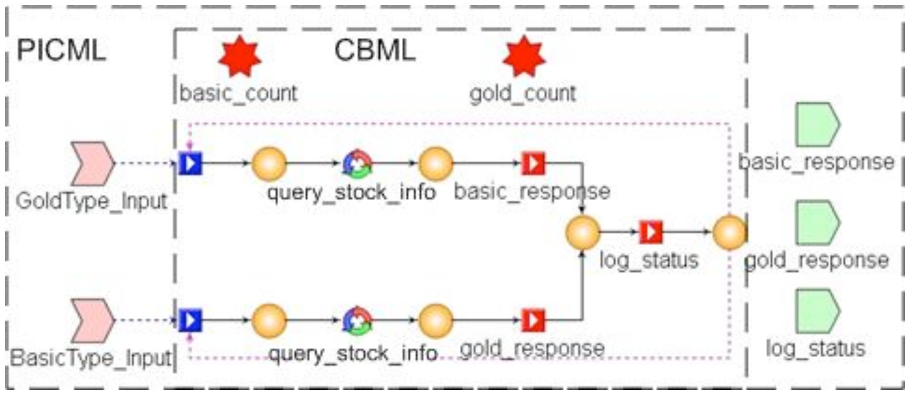


Figure III.12: Realization of integrating CBML and WML with PICML in CoSMIC

III.4 Preserving Formal Semantics of High-level Behavior Models

In Section III.3, we discussed how we integrated CBML with structural DSMLs (*e.g.*, PICML) to associate behavior models with structural models. In this section, we discuss how we use semantic anchoring [18, 19, 82] to preserve the formal semantics of CBML and generate configuration files for simulation tools based TIOA. We limit our discussion to the generation of configuration files for individual components, and not the entire system (*e.g.*, nodes, communication channels, etc.) because it is outside the scope of this dissertation.

III.4.1 Brief Overview of Semantic Anchoring

One of the main benefits of a DSML is its ability to allow developers to work with artifacts that are familiar to their domain. Although a DSML can help raise the level of abstraction – and simplify the development process by automating tedious and error-prone tasks – many DSMLs lack methods for proving their validity through formal specification of their semantics. Because it can be hard to formally define the semantics of a DSML in ways similar to formal mathematical languages such as I/O automata, Timed Automata, and Statecharts, it is becoming common practice to leverage semantic anchoring as a method to formally define the semantics of a DSML.

In semantic anchoring, developers rely on well-defined transformations that map elements of the DSML in question to elements of an existing formal language. This alleviates the necessity to formally define the semantics of a DSML because if the transformation functions are well-defined and the target language is semantically valid, then one can argue that the semantics of DSML in question is formally defined in the context of the target language. The remainder of this section discusses how we use semantic anchoring to map CBML to TIOA.

III.4.2 Transforming CBML into Timed I/O Automata

When we originally designed CBML, we based its definition on aspects from I/O automata because I/O automata possessed many of the same characteristics of components. In order to validate QoS from a simulation standpoint—as opposed to an emulation standpoint—we extended CBML to support timing so it would be consistent with TIOA. This would permit us to start understanding QoS properties such as end-to-end deadlines, service rates, and expected execution times, from a simulation perspective.

In TIOA, an automaton $\mathscr{A} = (\beta, I, O)$ is a tuple where

- $\beta = (X, Q, \Theta, E, H, \mathscr{D}, \mathscr{T})$ is a timed automata,
- I and O partition E into *input* and *output* actions, respectively.

We do not present the complete definition of B and its properties in this dissertation, and encourage the reader to see [57] for more details.

In order to leverage TIOA for semantic anchoring, we must first define a set of transformations that map CBML to TIOA. It is obvious that many of the elements in the definition of BM, which is used to formally define CBML, already occur in $\mathscr{A}$. Therefore, when we use the following transformation function:

$$trans(X_{BM}) \rightarrow Y_{\mathscr{A}}, \tag{III.8}$$

where X is an element in the definition of BM that is being transformed into element Y in the definition of $\mathscr{A}$, we define the following transformations:

$$trans(V_{BM}) \rightarrow X_{\mathscr{A}}, \tag{III.9}$$

$$trans(S_{BM}) \rightarrow Q_{\mathscr{A}}, \tag{III.10}$$

$$trans(\Theta_{BM}) \rightarrow \Theta_{\mathscr{A}}, \tag{III.11}$$

$$trans(I_{BM}) \rightarrow I_{\mathscr{A}}, \tag{III.12}$$

$$trans(O_{BM}) \rightarrow O_{\mathscr{A}}, \tag{III.13}$$

$$trans(A_{BM}) \rightarrow H_{\mathscr{A}}, \tag{III.14}$$

$$trans(T_{BM}) \rightarrow \mathscr{D}_{\mathscr{A}}, \tag{III.15}$$

$$trans(E_{BM}) \rightarrow s_{\mathscr{A}}, \tag{III.16}$$

where $s_{\mathscr{A}} \in Q_{\mathscr{A}}$.

To further illustrate the transformation, we have applied the transformation functions to a simplified version of the database component in the DSA illustrated in Figure III.13. The simplified version of the database component contains an input action named `Basic-Type_Input` and an internal action named (`query_stock_info`). It also contains a single output action named (`send_result`). When we apply Equation (III.9) through Equation (III.16) to the CBML model in Figure III.13, we produce the TIOA configuration file presented in Listing III.1.

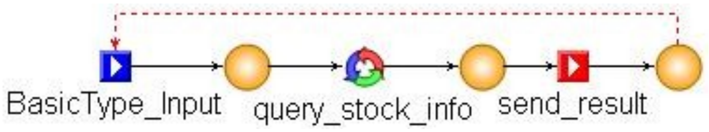


Figure III.13: Simplified version of the database component in the DSA

```
automaton DatabaseComponent (M: type)
  signature
    input BasicType_Input (m: M)
    internal handle_BasicType_Input
    internal query_stock_info
    output send_result (n: N)

  states
    next: Int := 1;
    queue_BasicType_Input : Seq[M] := {};
    clock : Int := 0;

  transitions
    input BasicType_Input (m)
      eff queue_BasicType_Input :=
        queue_BasicType_Input |- m;

    internal handle_BasicType_Input
      pre next = 1 /\
          queue_BasicType_Input ~= {};
      eff queue_BasicType_Input =
        tail (queue_BasicType_Input);
        next := 2;

    internal query_stock_info
      pre next = 2;
      eff next := 3; clock := clock + 10;

    output send_result
      pre next = 3;
      eff next := 1;

    trajectories
      trajdef thread
        evolve d(clock) = 1;
```

Listing III.1: Timed Input/Output Automaton configuration file for the simplified database component.

As shown in Listing III.1, the database component is converted to a single TIOA named `DatabaseComponent` that has a generic message type for receiving events. Each of the actions (*i.e.*, input, output, and internal) are converted to their equivalent TIOA element based on Equation (III.12), Equation (III.13) and Equation (III.14), respectively. Lastly, there is implicit corresponding internal action named handle_BasicInput_Event that is responsible for triggering the behavior sequence when an event is received on BasicInput_Event and placed in the corresponding event queue.

The DatabaseComponent automaton also contains three variables that hold the current state of the component (*i.e.*, $val(V)$). The next variable—which every component defines—determines what action to execute next in the behavior sequence since CBML sequences its behavior workflow. The queue_BasicType_Input is the event queue that stores events received on BasicType_Input. Each input action in CBML that is associated with an input event channel of a component always has an associated event queue. The clock variable is a continuous variable that represents time of execution. Its evolution is defined in the trajectories section of the automaton. Lastly, although the database component does not contain any explicit CBML variables, if the behavior model has any CBML variables, they are defined in the states section of the automaton.

Each of the CBML actions (*i.e.*, input, output, and internal) are converted to their respective TIOA elements. Because CBML forces a sequencing of the operations, we also defined TIOA preconditions (pre statements) and effects (eff statements) that will enforce this sequencing. As highlighted in Listing III.1, handle_BasicInput_Event is not enabled until BasicInput_Event has fired (*i.e.*, successfully executed its effects to change the automaton's state). Likewise, query_stock_info cannot fire until handle_BasicInput_Event fires and send_result cannot fire until query_stock_info fires. Because internal actions have a timing aspect associated with them, the effect of firing an internal actions also modifies the clock variable by the specified time, *i.e.*, Equation (III.5), to simulate the execution duration of the associated action. Lastly, it is clear that we only allow a single event to be active per behavior sequence, and not per component; however, we do not show this in the presented example.

III.4.3 Simulating Timed I/O Automata Models

Once CBML models are converted to TIOA models, developers can use them to define simulations that check different properties of individual components. The TIOA components we generate do not define complete simulations of the components because from a simulation standpoint, we do not know how developers want to follow different trajectories defined in the components. Instead, we generate the minimal sized component that allows developers to combine them with other TIOA models that define more trajectories, or simulations threads, to exercise the components. Moreover, since the models are converted to TIOA, developers can also leverage tools such as Tempo `(www.veromodo.com)`, which has tools and plug-ins to convert TIOA models to Timed Automata models, and other models types for theorem proving tools, thus satisfying Challenge 3 in Section III.1.

III.5 Code Generation Techniques for Facilitating Emulation

This section describes our approach for achieving code generation for emulation, which enables us to overcome serialized-phasing development and facilitate QoS validation during the early stages of the software lifecycle. Our current effort allows developers to generate emulation code for the CUTS, however, our code generation architecture is not dependent on CUTS (*e.g.*, Challenge 3 in Section III.1). Figure III.14 illustrates a conceptual model of our code generation architecture, which is composed from three technology independent, but programming language dependent layers of abstraction:

- **Emulation** – This layer represents the application layer's "business logic". The elements in WML used to parameterize the CBML behavior are mapped to this layer when model interpreters parse the model. For example, the *query_stock_info* action is generated at this layer in C++ code.

- **Templates** – This layer acts as a bridge [34] between the upper emulation layer and lower benchmarking layer. This allows both layers to evolve independently of

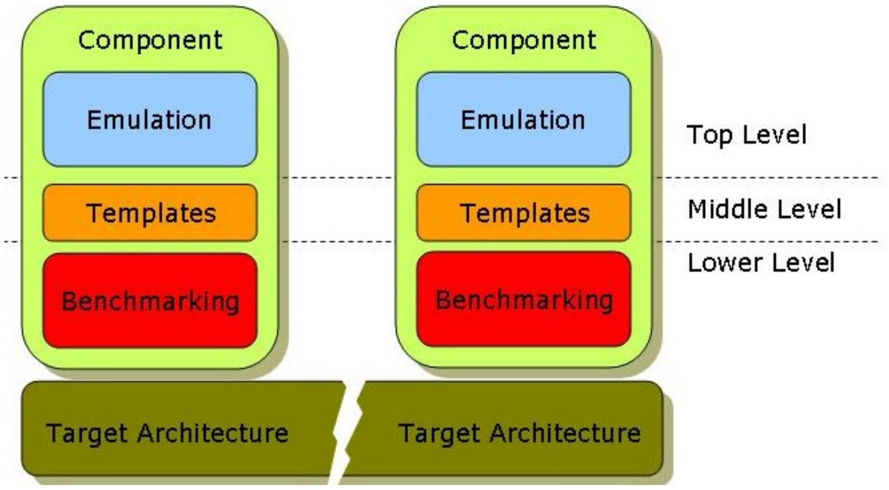


Figure III.14: Code generation architecture for emulation

each other. For example, if we want to provide support for other benchmarking frameworks we do not have to alter the generated code because the templates will provide the mapping. Likewise, if we ported the DSA to a different technology (or language) the code generator can tailor the source code to plug into this layer given we support the target programming language.

- **Benchmarking** – This layer represents the underlying benchmarking framework (*e.g.*, CUTS). Methods in this layer are invoked by the template layer above to capture workload metrics, such as execution timing of a database query by the database component, or response time of each user type in the context of the DSA.

Lastly, the encapsulating object for each of the three layers is the actual component hosted by the target architecture, which is language and technology dependent. The component is generated so that it has the same structure as its "real" counterpart (*e.g.*, same interfaces and attributes).

```
void DatabaseComponent::
push_BasicType_Input (QueryEvent *)
{
  // get activation record for this thread
  CUTS_Log_Record * record = CUTS_LOG_RECORD ();
  this->basic_count_ ++;

  record->perform_action_no_logging (
    CUTS_Database_Worker::
    query_stock_info (this->db_handle_));

  CUTS_CCM_Event_T <OBV_QueryResponse> __event_1__;
  this->ctx_->push_basic_response (__event_1__.in ());

  CUTS_CCM_Event_T <OBV_LogStatus> __event_2__;
  this->ctx_->push_log_status (__event_2__.in ());
}
```

Listing III.2: Excerpt of generated code from a PICML model extended with CBML and WML

Listing III.2 illustrates the generated code for a portion of the database component in the DSA. As illustrated in Listing III.2, the *push_BasicType_Input* method is the realization of implementing the *BasicType_Input* input event port in CCM. Each line of source code represents the WML actions used to parameterize the CBML behavior. The *record* is the template object that allows the emulation operations to be adapted for monitoring and analysis by benchmarking frameworks, such as CUTS.

III.6 Chapter Summary

This chapter described a model-driven engineering approach for overcoming serialized-phasing development of large-scale component-based DRE systems, and facilitate QoS evaluation continuously throughout the software lifecycle. Our approach defined two DSMLs, namely CBML and WML, that capture the behavior and workload of application components at a high-level. We then integrated these DSMLs with PICML, which models structural properties of component-based DRE systems. Lastly, we used model interpreters to tranform the behavior specifications to executable operations that leverage existing emulation frameworks, such as CUTS.

This approach also lays the foundation for facilitating continuous system integration and QoS validation of the large-scale component-based DRE systems, which is discussed in Chapter VI because as more is learned about the components, the behavior can be refined and regenerated for emulation. This implies that our approach is bridging the gap between agile developent methodologies and QoS evalation. Likewise, as the real application components are ready, they can replace the emulated components and their impact on system QoS can be observed. Lastly, we expect the results of real versus emulated components to match provided, the behavioral models of the emulated components approximate the real component behavior closely.

CHAPTER IV

GENERATIVE PROGRAMMING TECHNIQUES FOR ENABLING REUSE DOMAIN-SPECIFIC MODELING LANGUAGE MODEL INTERPRETERS

Component-based software engineering (CBSE) envisions the rapid development of large scale systems by assembling and deploying units of application functionality that is modularized into reusable components. Model-driving engineering (MDE) [108] is a dominant technology to realize the CBSE promise by automating the tasks of system specification, assembly, deployment and testing. Examples of MDE tools include (but are not limited to): Generic Modeling Environment (GME) [65], Eclipse Modeling Framework (EMF) [16], Domain-Specific Language Tools [39], openArchitectureWare [94], Matlab/-Simulink [123] and MetaEdit++ [124].

An important artifact of MDE frameworks are model interpreters, which provide generative capabilities to synthesize a variety of artifacts including deployment descriptors, interface descriptions, inputs to analysis tools, and testing code, among others. A common implementation technique used by model interpreters to implement the *interpreter logic* (*i.e.*, the logic to traverse the model hierarchy and generate the artifacts) is based on the Visitor [34] design pattern. The Visitor pattern separates the data structure (in our case the model) from the interpreter logic.

Despite the advantages of the Visitor design pattern, we have observed that contemporary approaches to implementing the model interpreters tightly couple both the traversal logic and generation logic thereby preventing any reuse when either or both of some of the traversal and generation requirements change. Furthermore, in component-based systems-of-systems, multiple modeling languages are often developed for individual subsystems that must be composed when these subsytems must be integrated. New design techniques are, therefore, needed to improve model interpreters implemented using the Visitor software design pattern to eliminate reinvention, as much as possible, and promote reuse of the interpreter logic.

Solution approach: Interpreter reuse via generative programming. A promising technique for promoting reuse in the core model interpreter logic is to use generative programming [24]. Generative programming is an attractive choice because it allows transparent alteration of core implementation (such as source code) on a per use case basis without affecting other entities that utilize the same core implementation.

This chapter describes *Metaprogrammable Interpreters for Model-driven Engineering (MIME)*, which is a metaprogrammable model interpretation technique to address the challenge of reinvention in model interpreters that use the Visitor pattern for generating different artifacts, and for composite DSMLs. First, we describe two contemporary approaches to developing model interpreters using the Visitor software design pattern and outline the reasons for reinvention manifested in these approaches. Second, we show how we can use templates to capture the correct visitation logic and generation logic of the target DSML. Lastly, we show how we use generative programming to generate interpreters that reuse (as opposed to reinvent) the core interpreter logic, but target different generated artifacts when implemented using the Visitor design pattern. Our experience using the MIME technique shows that we are able to implement model interpreters using the Visitor pattern that promote reuse of core interpreter logic but can be specialized on a per use case basis, such as suppressing or altering element visitation and varying generated artifacts.

Chapter organization. The remainder of this chapter is organized as follows: Section IV.1 discusses interpreter writing techniques wherein we illustrate the reasons for reinvention of the interpretation logic that use the Visitor pattern; Section IV.2 describes our generative programming solution for overcoming the reinvention required in existing solutions using the Visitor design pattern; and Section IV.3 summarizes the contributions of this chapter.

IV.1 Case Study: Emuluating Multiple Component-based Architectures

This section discusses current techniques for implementing MDE-based model interpreters using the Visitor design pattern. We discuss their limitations in the context of reinventing core interpreter logic.

IV.1.1 Commonality and Variability in Contemporary Component-based Architectures

Figure IV.1 illustrates a feature model highlighting several common artifacts of components from contemporary component-based architectures, such as CORBA Component Model (CCM), Enterprise Java Beans (EJB), and Microsoft.NET. The commonalities include a component, remote method invocation (RMI) ports, and event-based ports. The

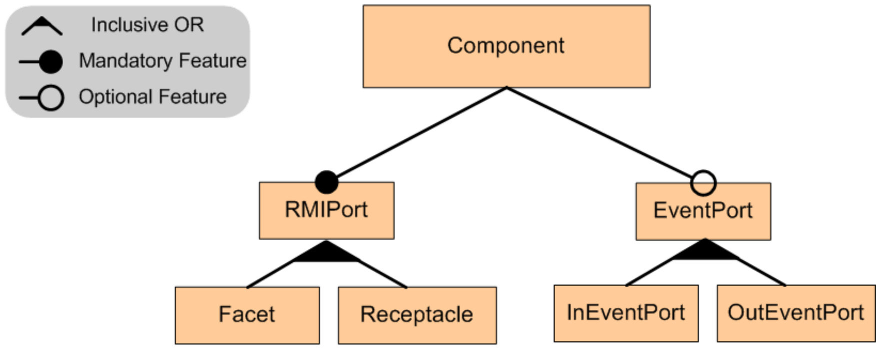


Figure IV.1: Subset of commonalities and variabilities in components from component-based architectures

variability is incurred along many dimensions: how the components are specified; the notion of a `EventPort` (*e.g.*, a port on the component in the context of CCM, a port to the Java Messaging Service (JMS) in the context of Java, or not applicable in Microsoft.NET); type of remote method invocation (*e.g.*, RMI in EJB versus IIOP in CCM); different programming languages (*e.g.*, C++ in the context of CCM, Java in the context of EJB, and C# in the context of Microsoft.NET); and many more not captured in Figure IV.1.

As explained in Chapter I, large-scale component-based DRE systems are often realized by integrating heterogeneous technologies, such as the aforementioned component-based architectures. DSML-based SEM tools used to assist with the development of large-scale component-based DRE systems—especially when emulating the system during the early stages of the software lifecycle to overcome serialized-phasing development—must support such characteristics. It is, therefore, desirable that model interpreters be able to provide maximal reuse for common artifacts across the different component-based architectures. At the same time, however, such model interpreters must be flexible enough to account for the variability of component-based architectures.

In the context of visitor-based model interpreters, we define the following terms, which are used throughout the remainder of this chapter:

- **Points-of-Visitation** – A point-of visitation defines how the model is traversed and how the model elements are visited.
- **Points-of-Generation** – A point-of-generation defines the code segment that generates specific artifacts for the visited model elements.
- **Interpreter Logic** – Interpreter logic is the entire interpreter code, which is the collection of points-of-visitation and points-of-generation.

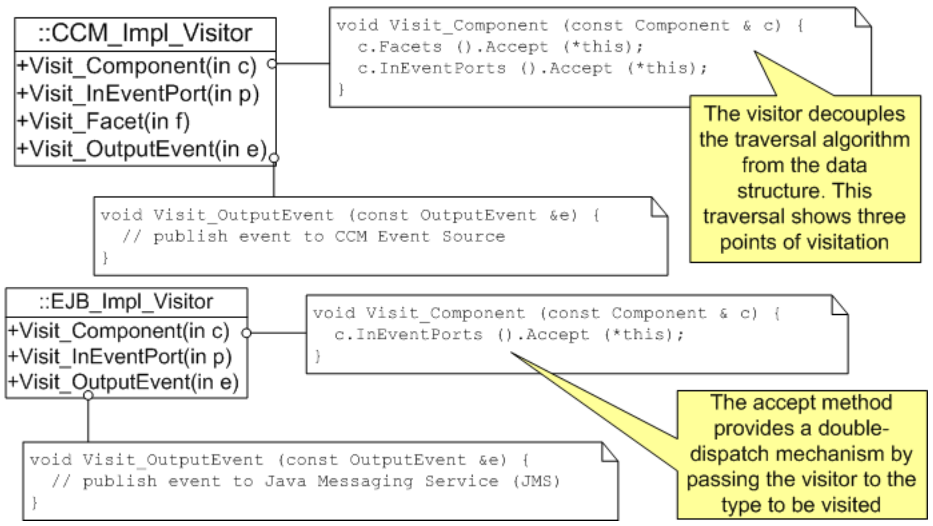
Note that the variability in the component-based architectures described above incurs corresponding variability in the points-of-visitation and points-of-generation in the interpreter logic. Our goal, therefore, is to develop the techniques to handle such variability while promoting reuse of interpreter logic.

IV.1.2 Inflexibility of Conventional Visitor-based Implementation Techniques

There are two primary techniques we have observed for implementing model interpreters using the Visitor design pattern. The first technique we denote as *single interpretation* and the second technique *strategized interpretation*.

IV.1.2.1 Single Interpretation

Single interpretation is the simplest—and sometimes the quickest—approach for writing model interpreters using the Visitor design pattern. In single interpretation, the core interpreter logic visits (or traverses) only elements of interest. When each element of interest is visited, the interpreter logic generates artifacts that correspond to that particular element.


::CCM_Impl_Visitor
+Visit_Component(in c)
+Visit_InEventPort(in p)
+Visit_Facet(in f)
+Visit_OutputEvent(in e)
The visitor decouples the traversal algorithm from the data structure. This traversal shows three points of visitation
::EJB_Impl_Visitor
+Visit_Component(in c)
+Visit_InEventPort(in p)
+Visit_OutputEvent(in e)
The accept method provides a double-dispatch mechanism by passing the visitor to the type to be visited

```
void Visit_Component (const Component & c) {
  c.Facets ().Accept (*this);
  c.InEventPorts ().Accept (*this);
}
```

```
void Visit_OutputEvent (const OutputEvent &e) {
  // publish event to CCM Event Source
}
```

```
void Visit_Component (const Component & c) {
  c.InEventPorts ().Accept (*this);
}
```

```
void Visit_OutputEvent (const OutputEvent &e) {
  // publish event to Java Messaging Service (JMS)
}
```

Figure IV.2: Conceptual model of single interpreter's implementation

As illustrated in Figure IV.2, we have defined two different interpreters, which generate source code for two different component-based architectures, namely `CCM_Impl_Visitor` and `EJB_Impl_Visitor`. Both implementations use the Visitor software design pattern for traversing models constructed using the modeling languages introduced in Chapter III. In Figure IV.2, the `CCM_Impl_Visitor` highlights 4 points-of-visitation (*i.e.*, `Visit_Component`, `Visit_InEventPort`, `Visit_Facet`, and `Visit_OutputEvent`) whereas `EJB_Impl_Visitor` highlights three points-of-visitation. Both interpreters have the same number of points-of-generation as the number of points-of-visitation. All visit methods (*i.e.*, `Visit_*` methods) both captured and not captured in Figure IV.2 represent reinvented interpreter logic in both implementations.

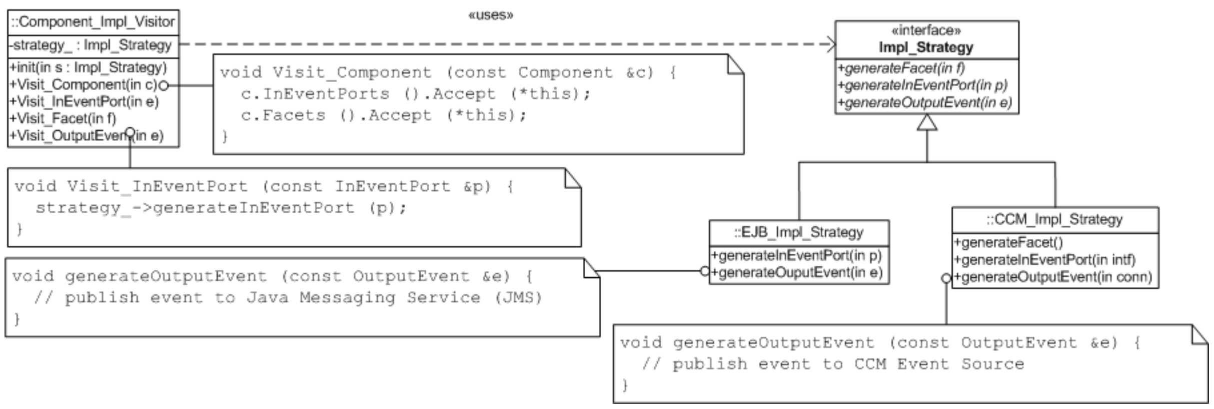
The main advantage of single interpretation model interpreters is that each interpreter contains efficient model traversing logic (within the limits of the Visitor design pattern) because it visits only elements of interest. The drawback of this approach is that each interpreter has to reinvent the core traversing logic only because it gets tangled with the generation logic.

IV.1.2.2 Strategized Interpretation

One approach to partially overcome the drawbacks of single interpretation is to use the Strategy [34] design pattern to implement a base class that contains points-of-generation as methods. The core interpreter logic is then implemented in terms of the base strategy class. Each model interpreter implements concrete classes derived from the base strategy that override the points-of-generation to generate the appropriate metadata. Lastly, the concrete class is used to strategize the core interpreter logic implemented in terms of the base strategy class.

As illustrated in Figure IV.3, the base class `Impl_Strategy` contains three points-of-generation: `generateFacet`, `generateInEventPort`, `generateOutputEvent`. Each concrete class derived from the base strategy class can override each point-of-generation to generate the appropriate artifact(s). If a point-of-generation is not overridden, then the default method—usually an empty method—is used. Similar to the single interpretation, the traversal logic is implemented using the Visitor pattern. The main difference is that instead of coupling the traversal logic with the generation logic, the traversal logic (*i.e.*, `Component_Impl_Visitor`) contains a base pointer to the appropriate strategy for the generation logic (*i.e.*, `Impl_Strategy`) which invokes the appropriate point-of-generation to generate the correct artifacts.

The advantage of strategized interpretation is that the traversal logic is decoupled from the generation logic, which permits reuse of the traversal logic. The drawback of this approach is that all points-of-generation are invoked for each strategy—even if it is not



```
void Visit_Component (const Component &c) {
  c.InEventPorts ().Accept (*this);
  c.Facets ().Accept (*this);
}
```

```
void Visit_InEventPort (const InEventPort &p) {
  strategy_->generateInEventPort (p);
}
```

```
void generateOutputEvent (const OutputEvent &e) {
  // publish event to Java Messaging Service (JMS)
}
```

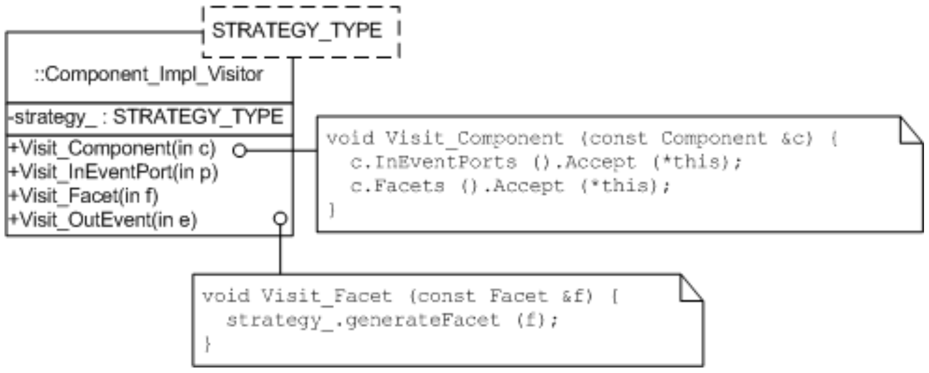
```
void generateOutputEvent (const OutputEvent &e) {
  // publish event to CCM Event Source
}
```

Figure IV.3: Conceptual model of a strategized interpreter's implementation

explicitly overridden—because the interpreter invokes methods on the base class and not the concrete class. As illustrated in Figure IV.3, `EJB_Impl_Strategy` does not override `generateFacet`; however, `Component_Impl_Visitor` is not aware that `EJB_-Impl_Strategy` ignores this method.

A special case of strategized interpretation is to use a template approach, which we call *parameterized strategy interpretation*. As illustrated in Figure IV.4, the main interpreter logic (*i.e.*, `Component_Impl_Visitor`) is written as a template class that is parameterizable by concrete types (*e.g.*, `CCM_Impl_Strategy` or `EJB_Impl_Strategy`). Similar to the strategized interpreter, the interpreter logic is implemented by invoking points-of-generation on the template class, *i.e.*, `STRATEGY_TYPE`, where the template class is implemented using the strategy design pattern.

The advantage of parameterized strategy interpretation is that each interpreter is capable of reusing the core interpreter logic and does not pay the penalty of invoking each point-of-generation. For example, `EJB_Impl_Strategy` is not concerned with the `generateFacet` point-of-generation and does not override this method. When `Compo-nent_Impl_Visitor` is parameterized with `EJB_Impl_Strategy`, it will use the default implementation for `generateFacet`—an empty method. Since `generateFacet`



```
void Visit_Component (const Component &c) {
  c.InEventPorts ().Accept (*this);
  c.Facets ().Accept (*this);
}
```

```
void Visit_Facet (const Facet &f) {
  strategy_.generateFacet (f);
}
```

Figure IV.4: Conceptual model of a template method interpreter's implementation

would be considered *dead code* [20], the C++ compiler will eliminate it from the executable as an optimization [131].

Although this approach is flexible in handling the variability in the points-of-generation, it is rigid with respect to the points-of-visitation. We seek a capability that allows us flexibility in both these artifacts while promoting maximal reuse.

IV.2 Enabling Reuse and Handling Variability in Model Interpreters

We now present our generative programming solution based on template metaprogramming [3] to promote reuse in model interpreters while addressing variability in the points-of-visitation and points-of-generation. Our solution is realized in a framework called MIME (Metaprogrammable Interpreters for Model-driven Engineering), which uses C++ template metaprogramming and facilities offered by the Boost library [55].

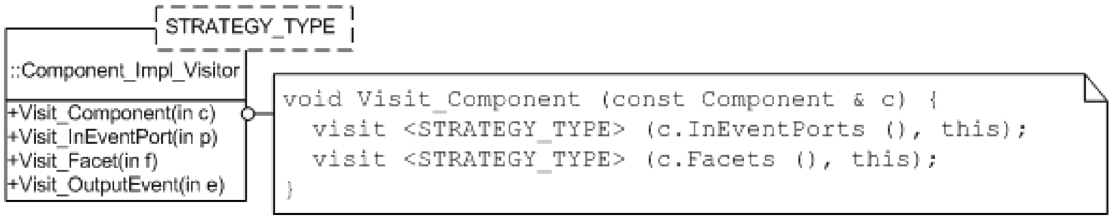
IV.2.1 Overview of Template-based Generative Programming

Generative programming is a style of computer programming that uses programs to write programs, or to transform existing programs. Several flavors of generative programming exist including, but not limited to, aspect-oriented programming [59], feature-oriented programming [11], and template metaprogramming in C++ [3].

Our solution approach focuses on leveraging template metaprogramming in C++ to enable reuse of model interpreter logic. In C++ template metaprogramming, applications rely on templates (and template specialization) to derive the final behavior of the application. More importantly, the derivation is done at compile-time enabling the compiler to evaluate and optimize the final executable as much as possible.

IV.2.2 Guidelines to Handle Variability in Points-of-Visitation and Generation

Figure IV.5 depicts the general guidelines to using template metaprogramming to handle the variability in the points-of-visitation and modularize them into reusable model interpreter code blocks. These guidelines enhance the parameterized strategy implementation illustrated in Figure IV.4.



```
void Visit_Component (const Component & c) {
  visit <STRATEGY_TYPE> (c.InEventPorts (), this);
  visit <STRATEGY_TYPE> (c.Facets (), this);
}
```

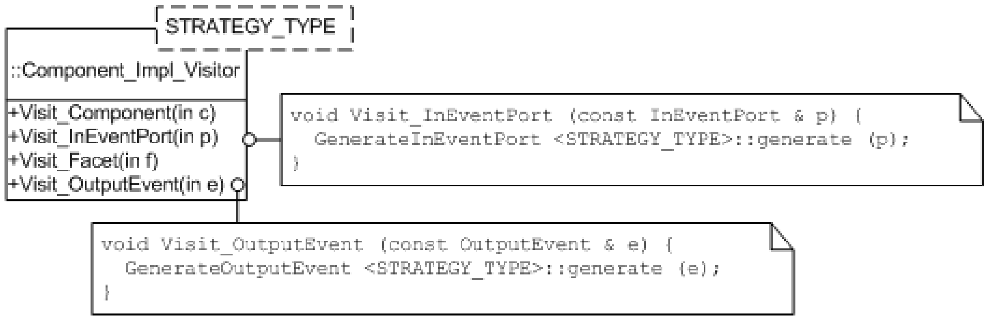
Figure IV.5: Handling variability in points-of-visitation

Specialization of the traversal logic is the ability to transparently customize how the model is traversed (visited) without actually modifying the existing overall interpreter logic. In order to promote specialization of the traversal logic, one must first determine the different points-of-visitation.

Once these are determined, as alluded to in Figure IV.5, each point-of-visitation must be wrapped by a parameterizable `visit` function. The `visit` function determines if the element of interest is visitable by the interpreter of `STRATEGY_TYPE` type. If the

STRATEGY_TYPE interpreter is interested in the specified element type, then the interpreter logic will visit it. Otherwise, the interpreter logic will ignore it (*i.e.*, remove its implementation at compile time).

Similar to how we used template metaprogramming to handle the variability in the points-of-visitation, we can also use it for points-of-generation. Figure IV.6 depicts the general guidelines for doing so. As highlighted in Figure IV.6, each point-of-generation is



```
void Visit_InEventPort (const InEventPort & p) {
  GenerateInEventPort <STRATEGY_TYPE>::generate (p);
}
```

```
void Visit_OutputEvent (const OutputEvent & e) {
  GenerateOutputEvent <STRATEGY_TYPE>::generate (e);
}
```

Figure IV.6: Handling variability in points-of-generation

an object that contains a static generate function. The point-of-generation object is parameterized by the STRATEGY_TYPE. If the point-of-generation is not specialized, then the default implementation of the point-of-generation—usually an empty method whose implementation is suppressed at compile time—is used; otherwise, the specialized implementation of STRATEGY_TYPE is used.

It is possible to use the parameterizable strategy interpreter technique discussed in Section IV.1.2 to implement the specialization of points-of-generation. By specifying point-of-generations in parameterizable objects, however, we can define traits that explicitly suppress points-of-generation, similar to how we handle points-of-visitation. This provides greater flexibility and control for customizing the generalized interpreter logic on a per interpreter basis.

IV.2.3 MIME Framework for Handling Variability in Points-of-Visitation and Generation

We now present how we have concretely realized the guidelines discussed in Section IV.2.2 in MIME using C++ template metaprogramming. Listing IV.1 highlights six main C++ templates we have developed for modularizing the traversal logic, *i.e.*, points-of-visitation.

```
// legend:
// S = interpreter; T = element(s); F = functor
struct NIL {
  static inline bool execute (...)
    {return false;}
};

template <typename S, typename T, typename F>
struct visit_all_t {
  static inline bool execute (T t, F f) {
    std::for_each (t.begin (), t.end (), f); return true;
  }
};

template <typename S, typename T, typename F>
struct visit_single_t {
  static inline bool execute (T t, F f) {
    f (t); return true;
  }
};

template <typename S, typename T>
struct visit_type {
  static const bool result_type = true;
};

template <typename S, typename T, typename F>
struct visit_t {
  typedef typename if_then_else <
    is_container <T>::result_type ,
    visit_all_t <S, T, F>,
    visit_single_t <S, T, F> >::result_type
    result_type;
};
```

```
35
36 template <typename S, typename T, typename F>
37 inline bool visit (T t, F f) {
38   typedef typename
39     get_type <T, is_container <T>::result_type >::
40        result_type type;
41
42   typedef typename
43     if_then_else <
44       visit_type <S, type::result_type >::result_type,
45       visit_t <S, T, F>::result_type,
46       NIL >::result_type result_type;
47
48   return result_type::execute (t, f);
49 }
```

Listing IV.1: Enabling transparent specialization of points-of-visitation.

- **`NIL`** (line 3) is the structure that suppresses the execution of a method at compile time. Because the method never has any side effects and the return value is known *a priori*, the compiler will use the return value directly when optimizations are enabled.

- **`visit_all_t`** (line 9) is an example iteration structure that is an adapter for the Standard Template Library `for_each` function located in the `<algorithm>` header file. It defines one static method called `execute`, which is needed so it can be used by `visit` (line 37).

- **`visit_single_t`** (line 16) is another example iteration structure that is used when interacting with non-container objects (*e.g.,* a single element). It also defines one static method named `execute`.

- **`visit_type`** (line 23) is a trait structure that determines if a particular element type `T` is visitable by interpreter `S`. Each interpreter can specialize `visit_type` to suppress visiting individual types, or all types. The default behavior is for each interpreter type to visit all element types.

- **`visit_t`** (line 28) is the trait that determines how to iterate over the element(s) in within object `t`, and what functor `f` to apply to `t`. The `visit_t` trait parameterizes the `if_then_else` metaprogrammable template [130] with a conditional expression named `is_container` (line 30) that determines if `t` is a container object. If `t` is a container object, then `result_type` is set to `visit_all_t`; otherwise, it is set to `visit_single_t`. Different interpreters can alter the iteration mechanisms by specializing this structure as needed.

- **`visit`** (line 37) is the factory function that creates the correct iteration type. This function parameterizes `if_then_else` with the `result_type` of `visit_type`, the iteration type, and the `NIL` type. If the element type is visitable, the final `result_type` will be the iteration type; otherwise, it will be the `NIL` type. The last step is to invoke the `execute` method on the derived `result_type`.

By using the above templates (and others not captured in Listing IV.1) and specializing them as needed, we are able to transparently alter the interpreter logic on a per interpreter basis at compile-time.

A similar approach is used to handle the variability in the points-of-generation in MIME. Listing IV.2 lists six primary C++ artifacts that allow us to transparently specialize the generation-logic of interpreters:

```
// legend:
// G = generator; Pn = parameter n
struct NIL {
  static inline bool generate (...)
    {return false;}
};

template <typename T>
struct is_nil {
  static const bool result_type = false;
};

template < >
struct is_nil <NIL> {
```

```
15    static const bool result_type = true;
16  };
17
18  // 0-parameter
19  template <typename G>
20  inline bool generate (void) {
21    typedef typename
22      if_then_else <is_nil <G>::result_type,
23      NIL, G>::result_type result_type;
24
25    return result_type::generate ();
26  }
27
28  // 1-parameter
29  template <typename G, typename P1>
30  inline bool generate (P1 p1) {
31    typedef typename
32      if_then_else <is_nil <G>::result_type,
33        NIL, G>::result_type result_type;
34
35    return result_type::generate (p1);
36  }
37
38  // 5-parameter
39  template <typename G, typename P1, typename P2,
40  typename P3, typename P4, typename P5>
41  inline bool generate (P1 p1, P2 p2, P3 p3,
42                        P4 p4, P5 p5) {
43    typedef typename
44      if_then_else <is_nil <G>::result_type,
45        NIL, G>::result_type result_type;
46
47    return
48      result_type::generate (p1, p2, p3, p4, p5);
49  };
```

Listing IV.2: Enabling transparent specialization of points-of-generation.

- **`NIL`** (line 3) is the structure that suppresses a point-of-generation at compile time. Because the method never has any side-effects and the return value is known *a priori*, the compiler will use the return value directly when optimizations are enabled.

- **`is_nil`** (line 9) is a trait template that determines if a type `T` is `NIL`. The default `result_type` is `true` (line 10). We specialized `is_nil` with NIL (line 14) so that when `is_nil` is parameterized with `NIL`, its `result_type` is `false`.

- **`generate`** (line 20, 30, & 41) is an overloaded function that inserts points-of-generation into interpreter logic. The function determines if the point-of-generation is supported (or suppressed). If the point-of-generation is supported, it invokes the generate method of `G`; otherwise, it invokes `generate` on the `NIL` object. The points-of-generation currently can handle zero to five parameters (*i.e.,* parameters passed to the generate functor). Extending the `generate` function to support more than five parameters simply requires adding more template parameters to support the extra parameters.

By using the above templates (and others not captured in Listing IV.2) and specializing them as needed, we are able to transparently alter the generation-logic on a per interpreter basis at compile-time.

IV.2.4 Using MIME in Component Implementation Visitor Example

To illustrate how we can use the template approach discussed in Section IV.2.3, we applied it to the component implementation visitor example presented in Section IV.1.2. Listing IV.3 contains a code snippet of the example that uses the template approach highlighted in Figure IV.5 to illustrate how the customization of the interpreter logic is achieved on a per model interpreter basis. For completeness we implemented a trivial `visit` function, which is illustrated in Listing IV.1. Because the `visit` function is parameterizable, developers may define more domain-specific visit functions using template specialization techniques. For example, an interpreter developer may implement a visit function that iterates over `InEventPort` elements in alphabetical order.

```
// Component_Impl_Visitor_T.cpp
// Demonstrates how reusable interpreter code can be
// developed using template metaprogramming.
```

```
 4  template <typename S>
 5  void Component_Impl_Visitor::
 6  Visit_Component (const Component & c) {
 7    visit <S> (c.InEventPorts (),
 8      boost::bind (&InEventPort::accept, _1,
 9                   boost::ref (*this)));
10
11    visit <S> (c.Facets (),
12      boost::bind (&Facet::accept, _1,
13                   boost::ref (*this)));
14  }
15
16  // EJB_Impl_Visitor.h.
17  // Shows template specialization to override default
18  // behavior. In this case we eliminate traversal to a
19  // Facet.
20  template < >
21  struct visit_type <EJB_Impl_Strategy, Facet> {
22    static const bool result_type = false;
23  };
24
25  // instantiate a specialized visitor.
26  typedef Component_Impl_Visitor <EJB_Impl_Strategy>
27    EJB_Impl_Visitor;
```

Listing IV.3: Specializing Points-of-visitation for Component Visitor Example.

As illustrated in Listing IV.3, we implemented `Component_Impl_Visitor` as a template class that is parameterizable by the concrete interpreter's type (*e.g.,* `CCM_Impl_Strategy` or `EJB_Impl_Strategy`). We then use the `visit` function to insert customizable points-of-visitation into the interpreter logic (*i.e.,* lines 7–13). By default, all elements will be visited by the interpreter logic. Since the `EJB_Impl_Strategy` interpreter does not visit `Facet` elements, we specialized the `visit_type` trait in `EJB_Impl_Strategy` header file (line 21), which is used by the `visit` function to determine if an element is visitable. This causes `Component_Impl_Visitor` interpreter logic to suppress that point-of-visitation when constructing the implementation of the `Visit_Component` method for `EJB_Impl_Strategy`.

A similar approach applies to the points-of-generation. Listing IV.4 contains a code snippet of our example that uses MIME's approach illustrated in Figure IV.6 to highlight how customization of points-of-generation is achieved on a per model interpreter basis. For completeness, Listing IV.2 defines the empty point-of-generation templates that are specialized in Listing IV.4.

```
// Component_DP_Visitor_T.cpp
// Shows how to leverage MIME's template metaprograms
// to develop reusable interpreters with modularized
// points-of-specialization.
template <typename S>
void Component_Impl_Visitor::
Visit_Facet (const Facet & f) {
  generate <S::GenerateFacet> (f);
}

template <typename S>
void Component_Impl_Visitor::
Visit_InEventPort (const InEventPort & p) {
  generate <S::GenerateInEventPort> (p);
}

// CCM_Impl_Visitor.h
// Shows how to specialize the points-of-generation
// for connection.
class CCM_Impl_Strategy {
  struct GenereateFacet {
    bool generate (const Facet & p) {
      // generate code to define the facet of a
      // component
      return true;
    }
  };

  struct GenerateInEventPort {
    bool generate (const InEventPort & conn) {
      // generate code to read event from event
      // source on component
      return true;
    }
  };
  // other methods not listed
};
```

```
38
39 // EJB_Impl_Visitor.h
40 class EJB_Impl_Strategy {
41 public:
42   // eliminate any generation for Port
43   typedef NIL GenerateFacet;
44
45   struct GenerateInEventPort {
46     bool generate (const InEventPort &p) {
47       // generate code to read event from the
48       // Java Messaging Service
49       return true;
50     }
51   };
52   // other methods not listed
53 };
```

Listing IV.4: Specializing Points-of-Generation for Component Visitor Example.

As illustrated in Listing IV.4, `Visit_Facet` (line 6) and `Visit_InEventPort` (line 12) are visitor methods that invoke their respective generation methods (*e.g.,* `GenerateFacet` and `GenerateInEventPort`). Both `CCM_Impl_Strategy` and `EJB_Impl_Strategy` implement the `GenerateInEventPort` point-of-generation (line 29 & 45, respectively). The `EJB_Impl_Strategy` interpreter does not visit `Facet` elements (see Listing IV.3, line 21), and, therefore, `EJB_Impl_Strategy` does not implement the `GenerateFacet` point-of-generation. On the other hand, `CCM_Impl_Strategy` specializes the `GenerateFacet` point-of-generation (line 21), which enables its interpreter to generate the necessary metadata for the `Facet` element.

Since we leveraged the MIME's points-of-visitation and points-of-generation template metaprogramming technique, we were able to reuse the interpreter logic with both example interpreters. Moreover, we were able to transparently specialize the interpreter logic of the `EJB_Impl_Strategy` model interpreter without affecting the interpreter logic of the `CCM_Impl_Strategy` model interpreter.

IV.2.5 Reuse within Composite Model Interpreters

Composite domain-specific modeling languages (DSMLs) are those created from one or more existing DSMLs. This allows the parent DSML (*i.e.,* the one created from multiple DSMLs) to use elements from the child DSML when constructing valid models. One of the main benefits of this modeling technique is reuse of existing modeling languages that capture a domain. This is very common when multiple subsystems need to be integrated to form systems-of-systems.

Although DSMLs can be composed from other DSMLs, the parent DSML interpreter is responsible for interpreting the child DSML. It is ideal for the parent DSML to reuse the interpreter logic of the child DSML. This would eliminate the need for parent DSML from having to reinvent the interpreter logic of the child DSML, which can be a complex task if the child DSML is complex.

In Section IV.1.2 we showed how current techniques for implementing DSML model interpreters using the Visitor pattern do not support reuse of interpreter logic for single DSMLs. Existing techniques for implementing DSML model interpreters, therefore, do not fully meet the needs of composite DSML model interpreters either. In Section IV.2.3, however, we showed how MIME can support interpreter logic reuse. Because MIME's implementation techniques allowed us to reuse the interpreter logic for single DSMLs, the interpreter logic of the child DSML can also be reused by the composite DSML.

IV.2.6 Guidelines for Using MIME

The template classes and traits we discussed only allow interpreters to customize visitation and generation code. Situations will arise that offer an opportunity to add customization to interpreter logic that the current templates can not handle. For example, one interpreter may want to visit the elements of a container in reverse order; whereas, another interpreter may want to set an upper bound on the number of elements visited in a container, or change the container type. To address this, we have developed simple guidelines for such situations:

1. Create a trait class or function with a name that reflects its purpose. This makes it easier to understand how it should be used.

2. Regardless of using a trait class or a template function, the first template parameter is the concrete interpreter's type. This allows the concrete interpreter to be the primary specialization artifact.

3. If there are any remaining template parameters, they are classified as properties that can influence the template's final value (or behavior) via template specialization.

IV.2.7 Enhancing the User-friendliness of MIME

One major challenge of template metaprogramming is comprehending its implementation. Model interpreters that leverage MIME's implementation technique will have vast amounts of template logic embedded throughout its implementation. Although we strive to make it easy to identify and understand the templates used by MIME, it may not appear as simple to novice developers.

To address this challenge, we believe it is possible to use high-level scripting languages to assist in generating the necessary skeleton code for different interpretation intentions. We make this conjecture because the points-of-visitation and points-of-generations are well defined. Moreover, any new metaprogrammable templates introduced into the core interpreter's logic using the generalization technique discussed in Section IV.2.6 will also be well-defined.

It may be feasible to define a high-level language with constructs that allow developers to define the general parsing logic for the modeling language. This high-level parsing logic would then be transformed into an interpreter that uses MIME's implementation technique. This is similar to how Java Emitter Templates behave. This would alleviate the need for understanding how to write templates for MIME, and make MIME's implementation technique more acceptable to developers who have little experience with template metaprogramming.

IV.3 Chapter Summary

In this chapter we presented a generative programming technique that used C++ template metaprogramming to promote reuse in Visitor design pattern-based DSML model interpreters. The main goal of our technique is to enable reuse of core interpreter logic so that different interpreters for the same DSML with the same interpretation goal do not have to reinvent it. To meet our desired objectives, we implemented several C++ templates class (and trait classes) that allow us to insert points-of-visitation and points-of-generation into the core interpreter logic. Consequently, each individual model interpreter can customize the interpreter logic to meet its needs while reusing the core interpreter logic.

Using our technique, not only are we able to reuse interpreter logic across multiple model interpreters, such as those that target diffenent component-based architecture middleware, using the same DSML, but we are also able to reuse it within composite DSML model interpreters. The parent DSML model interpreter does not have to worry about implementing the interpreter logic of the child DSML because it is provided for them (similar to a header file distributed with a shared or static C++ library). Furthermore, this allows users of a DSML to define their own customized interpreters because the interpreter logic for the DSML is already provided and they only need to specialize the default implementation to meet their needs.

CHAPTER V

UNIT TESTING TECHNIQUES FOR EVALUATING COMPONENT-BASED DRE SYSTEM QOS

Unit testing [49, 74] is a software validation technique that involves testing functional properties of software, such as setter/getter methods of a class. For large-scale component-based DRE systems, unit testing traditionally tests the "business-logic". As explained in Chapter I, QoS concerns (such as end-to-end response time) of large-scale component-based DRE systems typically occurs during the software integration phase since testing such concerns requires a complete system [26]. The separation of testing functional and QoS concerns in such systems also creates a disconnect between the two, as illustrated in Figure V.1, which can result in unforeseen consequences on project development, such as missed project deadlines due to failure to meet QoS requirements.

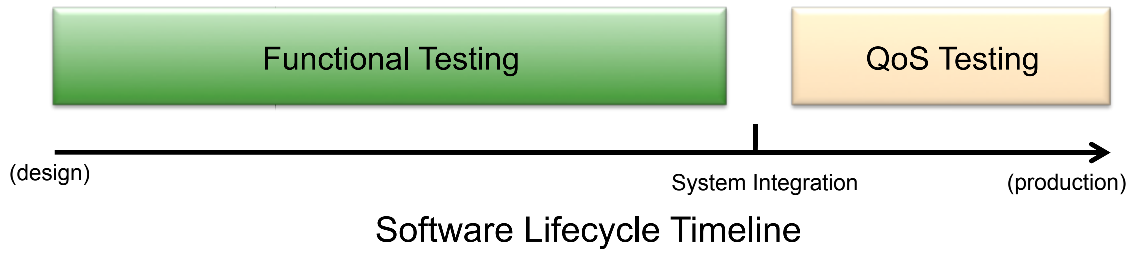


Figure V.1: Separation of functional and QoS testing in component-based distributed systems

As illustrated in Chapter III and Chapter IV, DSML-based SEM tools, such as CUTS, help bridge the gap between understanding how functional and QoS requirements affect each other. DSML-based SEM tools (1) model the behavior of a system under development and characterize its workload and (2) provide testers with artifacts to emulate the constructed models on the target architecture and analyze different QoS concerns, such as end-to-end response time. These tools enable system developers to evaluate QoS concerns prior to system integration (*i.e.*, before the system is completely developed).

DSML-based SEM tools, however, do not provide developers with techniques for unit testing QoS requirements of a component-based distributed system. Key QoS requirements include (1) identifying metrics of interest for data collection, (2) extracting such metrics for analysis, and (3) formulating equations based on extracted metrics that analyze an individual QoS concern or unit test. Moreover, in a development environment where requirements change constantly, system testers must also measure, evaluate, and reason about QoS concerns at the same level of abstraction as system requirements, and in a similar manner as functional concerns. Unfortunately, DSML-based SEM tools do not provide testers with techniques for abstracting and mapping low-level metrics to high-level system requirements and specifications. New techniques are therefore needed to unit test QoS concerns and reasoning about low-level system metrics at a higher-level of abstraction.

Solution Approach: High-level specification and analysis of QoS concerns. Evaluating QoS concerns of a component-based distributed system typically requires gathering metrics that are generated at different times while the system is executing in its target environment. For example, understanding a simple metric like system throughput requires capturing data that represents the number of events/users processed, the lifetime of the system, aggregating individual results, and calculating system throughput. For more complex metrics, the evaluation process becomes harder, particularly for large-scale component-based DRE systems or ultra-large-scale systems [51] composed of many components and deployed across many hosts.

This chapter describes a methodology and tool called *Understanding Non-functional Intentions via Testing and Experimentation (UNITE)* that is designed to alleviate the complexity of specifying and analyzing QoS concerns of component-based distributed systems. UNITE is based on relational database theory [8] where metrics used to evaluate a QoS unit test are associated with each other via relations to construct a data table. The constructed metric's table is evaluated by applying an SQL expression based on a user-defined function. Testers can use UNITE to evaluate QoS properties of their applications via the following steps:

1. Use log messages to capture metrics of interests, such as time an event was sent or values of elements in an event;

2. Identity metrics of interest within the log messages using message constructs, such as: `{STRING ident} sent message {INT eventId} at {INT time}`;

3. Define unit tests to analyze QoS concerns (such as overall latency, end-to-end response time, or system reliability and security) by formulating equations using the identified metrics, which can span many log messages.

Our experience applying UNITE to a representitive large-scale component-based DRE system shows it is an effective technique for unit testing QoS concerns during the early stages of system development. Moreover, as new concerns arise testers need only add new log message(s) to capture the metric(s) along with high-level construct(s) to extract the metric(s). UNITE thus significantly reduces the complexity of specifying QoS unit tests while producing a repository of historical data that can be analyzed and monitored throughout a distributed system's software development lifecycle.

Chapter Organization. The remainder of this chapter is organized as follows: Section V.1 summarizes a representative distributed system case study used to motivate the need for UNITE; Section V.2 describes the structure and functionality of UNITE; Section V.3 shows how we applied UNITE to our case study; and Section V.4 summarizes the contributions of this chapter.

V.1 Case Study: The QED Project

The Global Information Grid (GIG) middleware [2] is a large-scale component-based DRE system designed to ensure that different applications can collaborate effectively and deliver appropriate information to users in a timely, dependable, and secure manner. Due to the scale and complexity of the GIG, however, conventional implementations do not provide adequate end-to-end quality-of-service (QoS) assurance to applications that must respond rapidly to priority shifts and unfolding situations.

The QoS-Enabled Dissemination (QED) [69] project is a multi-organization collaboration designed to improve GIG middleware so it can meet QoS requirements of users and component-based distributed systems. QED's aims to provide reliable and real-time communication middleware that is resilient to the dynamically changing conditions of GIG environments. Figure V.2 shows QED in the context of the GIG. At the heart of the QED

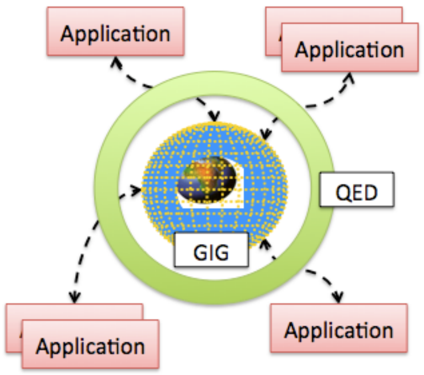


Figure V.2: Conceptual model of QED in the context of the GIG

middleware is a Java-based information broker based on the Java Messaging Service and JBoss that enables tailoring and prioritizing of information based on mission needs and importance, and responds rapidly to priority shifts and unfolding situations. Moreover, QED leverages technologies such as Mockets [126] and differentiated service queues [32] to provide QoS assurance to GIG applications.

The QED project is in its first year of development and is slated to run for several more years. Since the QED middleware is infrastructure software, applications that use it cannot be developed until the middleware itself is sufficiently mature. It is, therefore, hard for QED developers to ensure their software architecture and implementations are actually improving the QoS of applications that will ultimately run on the GIG middleware. The QED project thus faces a typical problem of serialized-phasing development as explained in Chapter I.

To overcome the serialized-phasing development problem, QED developers are using DSML-based SEM tools to execute performance tests against QED and continuously evaluate QoS properties throughout the software lifecycle. In particular, QED is using the CUTS SEM tool (see Chapter I). As explained in Chapter III, system testers use CUTS by modeling the behavior and workload of their component-based distributed system and, as explained in Chapter IV, use generative programming techniques to generating a test system for their target architecture using the constructed behavior and workload models. System testers then execute the test system on their target architecture, and CUTS collects performance metrics, which can be used to unit test non-functional concerns. This process is repeated continuously throughout the software development lifecycle of the system under development (see Chapter VI for more details).

In Chapter VI, we describe how integrating DSML-based SEM tools with continuous integtration environments provide a flexible solution for executing and managing component-based distributed system tests continuously throughout the development lifecycle. Before we can realize continuous system integration by integrating with continuous integration environments, we first need a methodology for presenting results to continuous integration environments, such as a QoS unit test. More importantly, we must solve the following challenges:

- **Challenge 1: Extracting data for metrics of interest.** Data extraction is the process of locating relevant information in a data source that can be used for analysis. In SEM tools, data extraction was limited to metrics that are known *a priori*, *e.g.*, at compilation time. It was therefore hard to identify, locate, and extract data for metrics of interest, especially if a QoS unit test needed data that not known *a priori*.

The QED testers needed a technique to identify metrics of interest that can be extracted from large amounts of system data. Moreover, the extraction technique should allow testers to identify key metrics at a high-level of abstraction and be flexible enough to handle data variation to apply CUTS effectively to large-scale component-based DRE systems.

- **Challenge 2: Analysis and aggregation extracted data.** Data analysis and aggregation is the process of evaluating extracted data based on a user-defined equation, and combining multiple results (if applicable) to a single result. This process is necessary since unit testing operates on a single result—either simple or complex—to determine whether it passes or fails. In conventional SEM tools, data analysis and aggregation is limited to functions known *a priori*, which made it hard to analyze extracted data via user-defined functions.

  The QED testers need a flexible technique for collecting metrics that can be used in user-defined functions to evaluate various system-wide non-functional concerns, such as relative server utilization or end-to-end response time for events with different priorities. Moreover, the technique should preserve data integrity (*i.e.*, ensuring data is associated with the execution trace that generated it), especially in absence of a globally unique identifier to identify the correct execution trace that generated it.

Due to these challenges, it was hard for system developers to use CUTS for conducting unit tests, which are executed by continuous integration environments, for QoS concerns of QED. Moreover, this problem extends beyond the QED project and applies to other large-scale component-based DRE systems that want to perform unit testing of QoS concerns. The remainder of this chapter shows how we addressed these challeges by improving CUTS so it can unit test QoS concerns of component-based distributed systems continuously throughout the software lifecycle.

V.2 UNITE: High-level Specification and Analysis of QoS Concerns

This section presents the underlying theory of UNITE and describes how it can be used to unit test QoS concerns of component-based distributed systems.

V.2.1 Specification and Extraction of Metrics from Text-based System Logs

System logs (or execution traces) are essential to understanding the behavior of a system, whether or not the system is distributed [54]. Such logs typically contain key data that can be used to analyze the system online and/or offline. For example, Listing V.1 shows a simple log produced by a system.

```
1 activating LoginComponent
2 ...
3 LoginComponent recv request 6 at 1234945638
4 validating username and password for request 6
5 username and password is valid
6 granting access at 1234945652 to request 6
7 ...
8 deactivating the LoginComponent
```

Listing V.1: Example log (or trace) produced by a system

As shown in Listing V.1, each line in the log represents a system effect that generated the log entry. Moreover, each line captures the state of the system when the entry was produced. For example, line 3 states when a login request was received by the `LoginComponent` and line 6 captures when access was granted to the client by the `LoginComponent`.

Although a system log contains key data to analyzing the system that produced it, the log is typically generated in a verbose format that can be understood by humans. This format implies that most data is discardable. Moreover, each entry is constructed from a well-defined format, which we call a *log format*, that will not change throughout the lifetime of the system execution. Instead, certain values (or variables) in each log format, such as time or event count, will change over the lifetime of the system. We formally define a log format $LF = (V)$ as:

- A set V of variables (or tags) that capture data of interest in a log message.

Moreover, Equation V.1 determines the set of variables in a given log format LF_i.

$$V = vars(LF_i) \tag{V.1}$$

Implementing log formats in UNITE. To realize log formats and Equation V.1 in UNITE, we use high-level constructs to identify variables $v \in V$ that contain data for analyzing the system. Users specify their message of interest and use placeholders—identified by brackets { }—to tag variables (or data) that can be extracted from an entry. Each placeholder represents variable portion of the message that may change over the course of the systems lifetime, thereby addressing Limitation 1 stated in Section V.1. Table V.1 lists

Table V.1: Log format variable types supported by UNITE

Type	Description
INT	Integer data type
STRING	String data type (with no spaces)
FLOAT	Floating-point data type

the different placeholder types currently supported by UNITE. Finally, UNITE caches the variables and converts the high-level construct into a regular expression. The regular expression is used during the analysis process (see Section V.2.3) to identify messages that have candidate data for variables V in log format LF.

```
LF1: {STRING owner} recv request {INT reqid} at {INT recv}
LF2: granting access at {INT reply} to request {INT reqid}
```

Listing V.2: Example log formats for tag metrics of interest

Listing V.2 exemplifies high-level constructs for two log entries from Listing V.1. The first log format (LF_1) is used to locate entries related to receiving a login request for a client (line 3 in Listing V.1). The second log format (LF_2) is used to locate entries related to granting access to a client's request (line 6 in Listing V.1). Overall, there are 5 tags in Listing V.2. Only two tags, however, capture metrics of interest: `recv` in LF_1 and `reply` in LF_2. The remaining three tags (*i.e.*, `owner`, `LF1.reqid`, and `LF2.reqid`) are used to preserve causality, which we explain in more detail in Section V.2.2.

V.2.2 Unit Test Specification for Analyzing QoS Concerns

Section V.2.1 disussed how we use log formats to identify entries in a log that contain data of interest. Each log format contains a set of tags, which are representative of variables and used to extract data from each format. In the trivial case, a single log format can be used to analyze a QoS concern. For example, if a developer wants to know how many events a component received per second, then the component could cache the necessary information internally and generate a single log message when the system is shutdown.

Although this approach is feasible, *i.e.*, caching data and generating a single message, it is not practical in a component-based distributed system because individual data points used to analyze the system can be generated by different components. Moreover, data points can be generated from components deployed on different hosts. Instead, what is needed is the ability to generate independent log messages and specify how to associate the messages with each other to preserve data integrity.

In the context of unit testing QoS concerns, we formally define a unit test $UT = (LF, CR, f)$ as:

- a set LF of log formats that contain variables V identifying which data to extract from log messages,
- a set CR of causal relations that specify the order of occurrence for each log format such that $CR_{i,j}$ means $LF_i \rightarrow LF_j$, or LF_i occurs before LF_j, and

- a user-defined evaluation function f based on the variables in LF.

Causal relations are traditionally based on time. UNITE, however, uses log format variables to resolve causality because it alleviates the need for a globally unique identifier to associate metrics (or data). Instead, one only needs to ensure that two unique log formats can be associated with each other, and each log format is in at least one causal relation (or association). UNITE does not permit circular relations because it requires human feedback to determine where the relation chain between log formats begins and ends.

We formally define a causal relation $CR_{i,j} = (C_i, E_j)$ as:

- a set $C_i \subseteq vars(LF_i)$ of variables that define the key to represent the cause of the relation, and
- a set $E_j \subseteq vars(LF_j)$ of variables that define the key to represent the effect of the relation.

Moreover, $|C_i| = |E_j|$ and the type of each variable (see Table V.1), *i.e.*, $type(v)$, in C_i, E_j is governed by Equation V.2:

$$type(C_{i_n}) = type(E_{j_n}) \tag{V.2}$$

where $C_{i_n} \in C_i$ and $E_{j_n} \in E_j$.

Implementing unit tests in UNITE. In UNITE, users define unit tests by selecting what log formats should be used to extract data from message logs. If a unit test has more than one log format, then users must create a causal relation between each log format. When specifying casual relations, users select variables from the corresponding log format that represent the cause and effect. Last, users define an evaluation function based on the variables in selected log formats.

For example, if a QED developer wants to create a unit test to calculate the duration of the login operation, then a unit test is created using LF_1 and LF_2 from Listing V.2. Next, a causal relation is defined between LF_1 and LF_2 as:

$$LF_1.reqid = LF_2.reqid. \quad \text{(V.3)}$$

Finally, the evaluation function is defined as:

$$LF_2.reply - LF_1.recv. \quad \text{(V.4)}$$

The following section discusses how we evaluate the function of QoS unit tests for component-based distributed systems.

V.2.3 Evaluating QoS Unit Tests

Section V.2.1 discussed how log formats can be used to identify messages that contains data of interest. Section V.2.2 discussed how to use log formats and casual relations to specify unit test for QoS concerns. The final phase of the process is evaluating the unit test, *i.e.*, the evaluation function f. Before we explain the algorithm used to evaluate a unit test's function, we must first understand different types of causal relations that can occur in a component-based distributed system.

As shown in Figure V.3, there are four types of causal relations that can occur in a component-based distributed system, which affect the algorithm used to evaluate a unit test. The first type (a) is one-to-one relation, which is the most trivial type to resolve

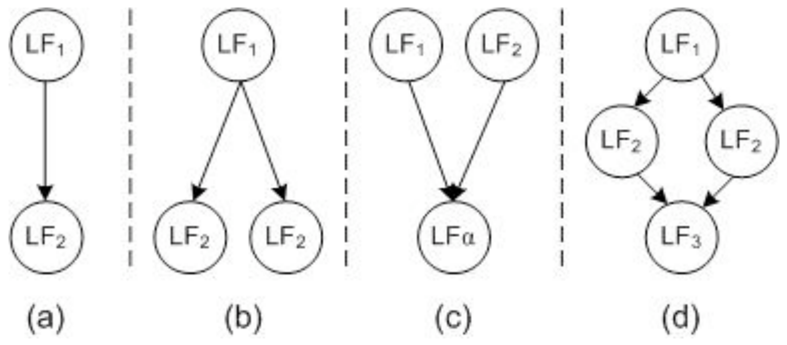


Figure V.3: Four types of causal relations that can occur in a component-based DRE system

between multiple log formats. The second type (b) is one-to-many relation and is a result of a multicast event. The third type (c) is many-to-one, which occurs when many different components send an event type to a single component. The final type (d) is a combination of previous types (a) – (d), and is the most complex relation to resolve between multiple log formats.

If we assume that each entry in a message log contains its origin (*e.g.*, hostname) then we can use a dynamic programming algorithm and relational database theory techniques to reconstruct the data table of values for a unit test's variables. Algorithm 1 lists our algorithm for evaluating a unit test. As shown in Algorithm 1, we evaluate a unit test UT given the collected log messages LM of the system. The first step to the evaluation process is to create a directed graph G where log formats LF are nodes and the casual relations $CR_{i,j}$ are edges. We then topologically sort the directed graph so we know the order to process each log format. This step is necessary because when causal relation types (b) – (d) are in the unit test specification, processing the log formats in reverse order of occurrence reduces algorithm complexity for constructing data set DS. Moreover, it ensures we have rows in the data set to accommodate the data from log formats that occur prior to the current log format. After topologically sorting the log formats, we construct a data set DS, which is a table that has a column for each variable in the log formats of the unit test.

Likewise, we sort the log messages by origin and time to ensure we have the correct message sequence for each origin. This step is necessary if one wants to see the data trend over the lifetime of the system before aggregating the results, which we discuss later in this chapter. Once we have sorted the log messages, we match each log format in LF' against each log message in LM'. If there is a match, then we extract values of each variable from the log message, and update the data set. If there is a cause variable set C_i for the log format LF_i, we locate all the rows in the data set where the values of C_i equal the values of E_j, which are set by processing the previous log format. If there is no cause variable set, we append the values from the log message to the end of the data set. Finally, we purge

Algorithm 1 General algorithm evaluating a unit test via log formats and causal relations

```
1: procedure EVALUATE($UT, LM$)
2:     $UT$: unit test to evaluate
3:     $LM$: set of log messages with data
4:     $G \leftarrow directed\_graph(UT)$
5:     $LF' \leftarrow topological\_sort(G)$
6:     $DS \leftarrow variable\_table(UT)$
7:     $LM' \leftarrow$ sort $LM$ ascending by (origin, time)
8:
9:     for all $LF_i \in LF'$ do
10:        $K \leftarrow C_i$ from $CR_{i,j}$
11:
12:        for all $LM_i \in LM'$ do
13:            if $matches(LF_i, LM_i)$ then
14:                $V' \leftarrow$ values of variables in $LM_i$
15:
16:                if $K \neq \emptyset$ then
17:                    $R \leftarrow findrows(DS, K, V')$
18:                    $update(R, V')$
19:                else
20:                    $append(DS, V')$
21:                end if
22:            end if
23:        end for
24:    end for
25:
26:    $DS' \leftarrow$ purge incomplete rows from $DS$
27:    return $f(DS')$ where $f$ is evaluation function for $UT$
28: end procedure
```

all the incomplete rows from the data set and evaluate the data set using the user-defined evaluation function for the unit test.

Handling duplicate data entries. For long running systems, it is not uncommon to see variations of the same log message within the complete set of log messages. Moreover, we defined log formats on a unit test to identify variable portions of a message (see Section V.2.1). We therefore expect to encounter the same log format multiple times. When constructing the data set in Algorithm 1, different variations of the same log format will create multiple rows in the final data set. A unit test, however, operates on a single value, and not multiple values. To address this concern, we use the following techniques:

- **Aggregation** – A function used to convert a data set to a single value. Examples of an aggregation function are, but not limited to: AVERAGE, MIN, MAX, and SUM.

- **Grouping** – Given an aggregation function, grouping is used to identify data sets that should be treated independent of each other. For example, in the case of causal relation (d) in Figure V.3, the values in the data set for each sender (*i.e.*, LF_2) could be considered a group and analyzed independently.

We require specifying an aggregation function as part of the evaluation equation f for a unit test because it is known *a priori* if a unit test may produce a data set with multiple values. We formally define a unit test with groupings $UT' = (UT, \Gamma)$ as:

- a unit test UT for evaluating a QoS concern, and

- a set $\Gamma \subseteq vars(UT)$ of variables from the log formats in the unit test.

Evaluating unit tests in UNITE. In UNITE, we implemented Algorithm 1 using the SQLite relational database (`sqlite.org`). To construct the variable table, we first insert the data values for the first log format directly into the table since it has no causal relations. For the remaining log formats, we transform its causal relation(s) into a SQL `UPDATE` query. This allows us to update only rows in the table where the relation equals values of interest in the current log message. Table V.2 shows the variable table constructed by UNITE for the example unit test in Section V.2.2. After the variable data table is constructed, we use the evaluation function and groupings for the unit test to create the final SQL query that evaluates the unit test.

```
SELECT AVG(LF2_reply - LF1_recv) AS result
  FROM vtable123;
```

Listing V.3: SQL query for calculation average login duration

Table V.2: Example data set produced from log messages

LF1_reqid	LF1_recv	LF2_reqid	LF2_reply
6	1234945638	6	1234945652
7	1234945690	7	1234945705
8	1234945730	8	1234945750

Listing V.3 shows Equation V.4 as an SQL query, which is used to evaluate the data set in Table V.2. The final result of this example—and the unit test—would be 16.33 msec.

V.3 Applying UNITE to the QED Project

This section analyzes results of experiments that evaluate how UNITE can address key testing challenges of the QED project described in Section V.1.

V.3.1 Experiment Setup

As mentioned in Section V.1, the QED project is in its first year of development and is expected to continue for several years. QED developers do not want to wait until system integration time to validate the performance of their middleware infrastructure relative to stated QoS requirements. QED testers therefore used CUTS and UNITE to perform early intergration testing. All tests were run in a representative testbed based on ISISlab (see Appendix A) configured with the Fedora Core 6 operating system.

To test the QED middleware, we first constructed several scenarios using CUTS' modeling languages (see Chapter III). Each scenario was designed such that all components communicate with each other using a single server in the GIG (similar to Figure V.2 in Section V.1). The first scenario was designed to test different thresholds of the underlying GIG middleware to pinpoint potential areas that could be improved by the QED middleware. The second scenario was more complex and emulated a *multi-stage workflow*. The multi-stage workflow is designed to test the underlying middleware's ability to ensure application-level QoS properties, such as reliability and end-to-end response time when handling applications with different priorities and priviledges.

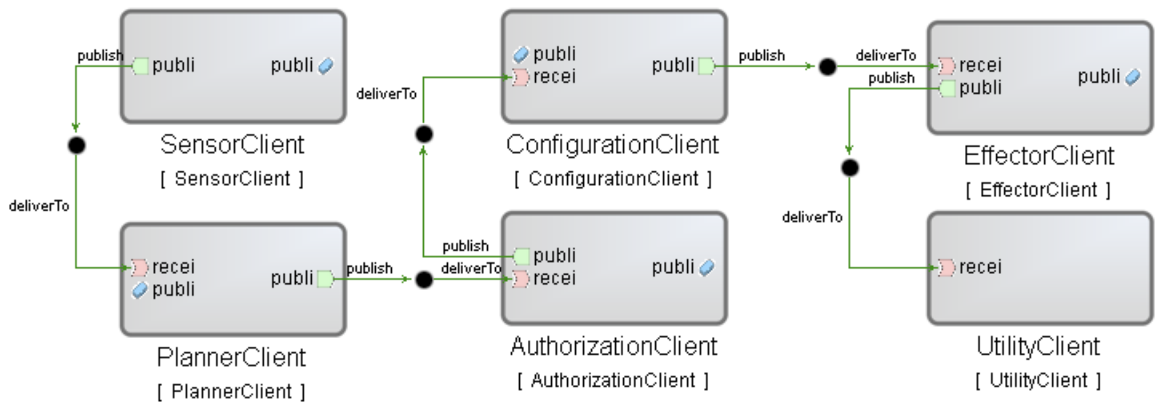


Figure V.4: CUTS model of the multi-stage workflow test scenario

As shown in Figure V.4, the multi-stage workflow has six types of components. Each directed line that connects a component represents a communication event (or stage) that must pass through the GIG (and QED) middleware before being delivered to the component on the opposite end. Moreover, each directed line conceptually represents where QED will be applied to ensure QoS between communicating components. Each component in the multi-stage workflow has a behavior model (see Chapter III) that dictates its actions during a test. Moreover, each behavior model contains actions for logging key data needed to evaluate a unit test, similar to Listing V.1 in Section V.2.1.

Listing V.4 lists example log messages from the multi-stage workflow scenario.

```
. MainAssembly.SurveillanceClient: Event 0: Published a
  SurveillanceMio at 1219789376684
. MainAssembly.SurveillanceClient: Event 1: Time to
  publish a SurveillanceMio at 1219789376685
```

Listing V.4: Example log messages from the multi-stage workflow scenario

These log message contain information about the event, such as the event id and the timestamp. Each component also generates log messages about the events it receives and its state (such as its event count). In addition, each component sends enough information to create a causal relation between itself and the receiver, so there is no need for a global unique identifier to correlate data.

After contructing the behavior and workload model for each component in the multi-stage workflow, we generated its source code such that it conforms to the GIG middleware using the techniques presented in Chapter IV. We next used UNITE to construct log formats (see Section V.2.1) for identifing log messages during a test run that contain metrics of interest. These log formats are also used to define unit tests that evaluate QoS concerns described in Section V.2.2. Overall, we are interested in evaluating the following concerns in our experiments:

- **Multiple publishers.** At any point in time, the GIG will have many components publishing and receiving events simultaneously. We therefore need to evaluate the response time of events under such operating conditions. Moreover, QED needs to ensure QoS when the infrastructure servers must manage many events. Since the QED project is still in the early stages of development, we must first understand the current capabilities of the GIG middleware without QED in place. These results provide a baseline for evaluating the extent to which the QED middleware capabilities improve application-level QoS.

- **Time spent in server.** One way to ensure high QoS for events is to reduce the time an event spends in a server. Since the GIG middleware is provided by a third-party vender, we cannot ensure it will generate log messages that can be used to calculate how long it takes the server to process an event. Instead, we must rely on messages generated from distributed application components whenever it publishes/sends an event.

For an event that propogates through the system, we use Equation V.5 to calculate how much time the event spends in the server assuming event transmission is instantenous, *i.e.*, negligible.

$$(end_e - start_e) - \sum_{c} S_{c_e} \tag{V.5}$$

As listed in Equation V.5, we calculate the time spent in the server by taking the response time of the event e, and subtracting the sum of the service times of the event in each component S_{c_e}. Again, since QED is in its early stages of development, this unit test provides a baseline of the infrastructure and used continuously throughout development.

V.3.2 Experiment Results

We now present results for experiments of the scenarios discussed in Section V.3.1. Since QED is still in the early stages of development, we focus our experiments on evaluating the current state of the GIG middleware infrastructure, *i.e.*, without measuring the impact of QED middleware on QoS, and UNITE's ability to unit test non-functional concerns.

Analyzing multiple publisher results. Table V.3 shows the results for the unit test that evaluates average end-to-end response time for an event when each publisher publishes at 75 Hz. As expected, the response time for each importance value is similar. When we unit tested this scenario using UNITE, the test results presented in Table V.3 were calculated from two different log formats—the log format generated by a publisher and the subscriber. The total number of log messages generated during the course of the test was 993,493.

UNITE also allows us to view the data trend for the unit test of this scenario to get a more detailed understanding of performance. Figure V.5 shows how the response time of the event increases over the lifetime of the experiment. We knew beforehand that this configuration for the unit test would produce too much workload. UNITE's data trend

Table V.3: Average end-to-end (E2E) response time (RT) for multiple publishers sending events at 75 Hz

Publisher Name	Importance	Avg. E2E RT (msec)
ClientA	30	103931.14
ClientB	15	103885.47
ClientC	1	103938.33

and visualazation capabilities, however, helped make it clear the extent to which the GIG middleware was being over utilized.

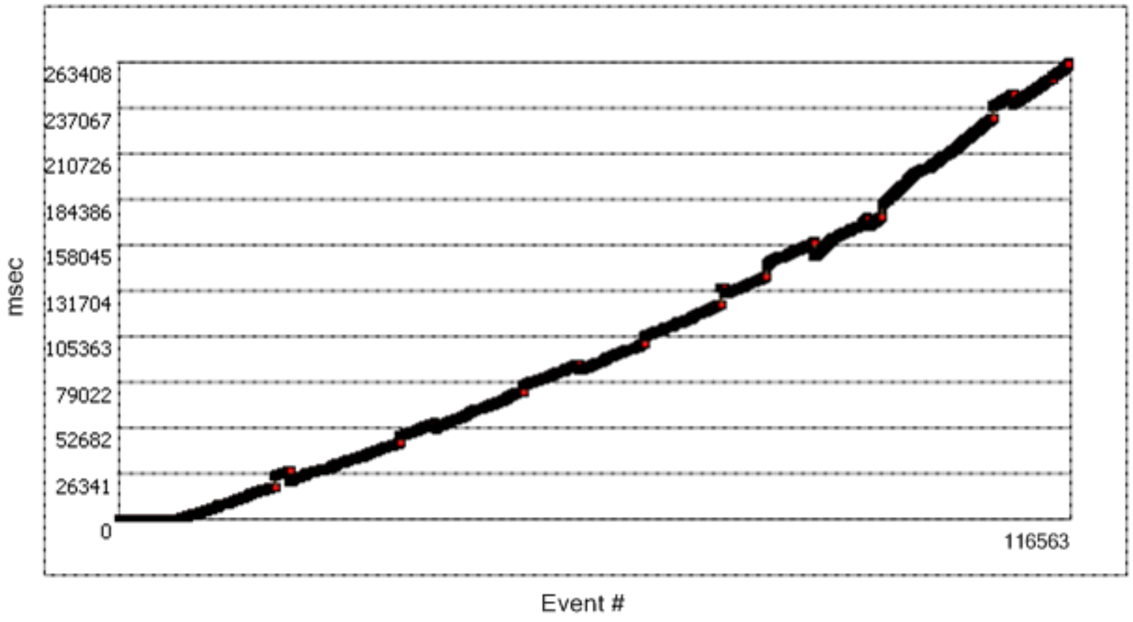


Figure V.5: Data trend graph of average end-to-end response time for multiple publishers sending events at 75 Hz

Analyzing maximum sustainable publish rate results. We used the multi-stage workflow to describe a complex scenario to test the limits of the GIG middleware without forcing it into incremental queueing of events. Figure V.6 graphs the data trend for the unit test, which is calculated by specifying Equation V.5 as the evaluation for the unit test, and was produced by UNITE after analyzing (*i.e.*, identifying and extracting metrics from) 193,464 log messages. The unit test also consisted of ten different log formats and nine different causal relations, which were of types (a) and (b), discussed in Section V.2.3.

Figure V.6 illustrates the sustainable publish rate of the mutle-stage workflow in ISISlab. As illustrated, the just-in-time compiler (JITC) and other Java features cause the middleware to temporarily increase the individual message end-to-end response time. By the end of the test (which is not shown in the Figure V.6), the time an event spends in the server reduces to normal operating conditions.

The multi-stage workflow results provides two insights to QED developers. First, their theory of the maximum publish rate in ISISlab is confirmed. Second, Figure V.6 helps developers speculate on what features of the GIG middleware might cause performance bottlenecks, how QED could address such problems, and what new unit test are needed to illustrate QED's improvements to the GIG middleware. By providing QED testers comprehensive testing and analysis features, UNITE helps guide the development team to the next phase of testing and integration of feature sets.

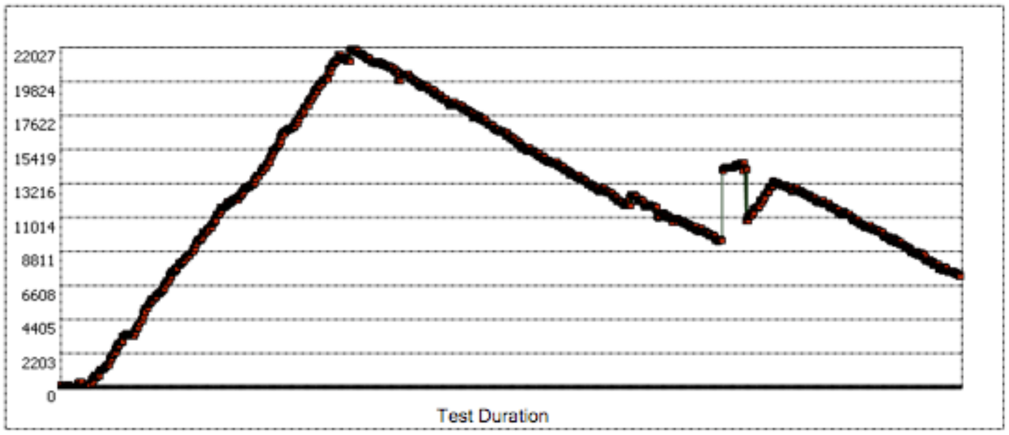


Figure V.6: Data trend of the system placed in near optimal publish rate

Evaluating the impact of UNITE on the experiment. Because of UNITE, we can quickly construct unit tests to evaluate the GIG middleware. In the context of the multi-stage workfow scenario, UNITE provids two insights to QED developers. First, their theory of the maximum publish rate in ISISlab is confirmed. Second, the data trend and the visualization capabilities of UNITE helped developers speculate on what features of the GIG middleware might cause performance bottlenecks, how QED could address such problems, and what new unit test are need to illustrate QED's improvements to the GIG middleware.

In addition, UNITE's analyical capabilities are not bounded by a unit test's configuration and data. As long as the correct log formats and their causal relations are specified, UNITE can evaluate the unit test. Moreover, we do not need to specify a global unique identify to associate data with its correct exeuction trace. If we require a global unique identifier to associate data metrics, then we have to ensure that all components propogate the identifer. Moreover, if we add new compoents to the multi-stage workflow, each component would have to be aware of the global unique idenfier, which can inherently complicate the logging specification.

By providing developers comprehensive testing and analysis capabilities, UNITE helps guide the QED developers to the next phase of testing and integration of feature sets. We therefore conclude that UNITE helped reduce the complexity of evaluating QoS concerns of a component-based distributed system. Moreover, unit tests can be automated and run continuously throughout the software lifecycle of QED using continuous integration environments, such as CruiseControl (`cruisecontrol.sourceforge.net`), as explained in Chapter VI.

V.4 Chapter Summary

QoS concerns of large-scale component-based DRE systems have traditionally been tested during integration. The earlier that QoS concerns are tested in the target environment, however, the greater chance of locating problematic areas in the software system [134, 138]. This chapter describes and evaluates a technique called *Understanding Non-functional Intentions via Testing and Experimentation (UNITE)* for unit testing QoS concerns of large-scale component-based DRE systems. UNITE uses log messages generated from a testing environment and message constructs that identify messages of interest. Moreover, testers define functions in terms of variable data in log messages that are used to evaluate QoS concerns of the system under development.

Our experience applying UNITE to a representative large-scale component-based DRE system shows how it simplifies identifying and extracting of metrics of interest for analyzing QoS concerns. Furthermore, the methodologies realized in UNITE helps to further bridge the gap between agile development methodologies and QoS evaluation experienced during serialized-phasing development of large-scale component-based DRE systems.

CHAPTER VI

TECHNIQUES FOR FACILITATING CONTINUOUS SYSTEM INTEGRATION TESTING

In Chapter I, we discussed how large-scale component-based DRE systems, such as air traffic control systems, mission avionics systems, shipboard computing environments, and traffic management systems, are traditionally affected by serialized-phasing development. Likewise, in Chapter III and Chapter IV, we showed how DSML-based SEM tools, such as CUTS, help overcome serialized-phasing development and bridge the gap between agile development and QoS evalation for such systems.

Although DSML-based SEM tools assist with overcoming serialized-phasing development experienced by large-scale component-based DRE systems, SEM tools have historically focused on performance analysis [113] rather than efficient testing (as currently shown throughout this dissertation). Moreover, due to the inherit complexities of serialized-phasing development (*e.g.*, portions of the system being developed in phases defer QoS testing until integration testing) it is hard to use DSML-based SEM tools for evaluating QoS continuously throughout software lifecycle. New techniques are therefore needed to enhance the capabilities of existing DSML-based SEM tools, and improve their capabilities for testing continuously throughout the software lifecycle.

Solution approach: Integrate SEM tools with continuous integration environments. Continuous integration [33] environments, such as CruiseControl.NET, Build Forge, and DART, continuously exercise the build cycle of software to validate its quality by (1) performing automated system builds upon source code check in or successful execution and evaluation of prior events, (2) executing suites of unit tests to validate basic system functionality, (3) evaluating source code to ensure it meets coding standards, and (4) executing code coverage analysis. Continuous integration environments, however, have been traditional used to evaluate functional concerns of large-scale component-based DRE systems since it is a conventional agile development technique (see Chapter I).

This chapter describes the research and design of *CiCUTS*, which combines the continuous integration environments with DSML-based SEM tools, such as CUTS. CiCUTS tests are deployed into the target domain to enable system QoS testing. These tests are managed by the continuous integration environment and dictate the behavior of CUTS, as shown in Figure VI.1. This process is repeated continuously to help developers and testers en-

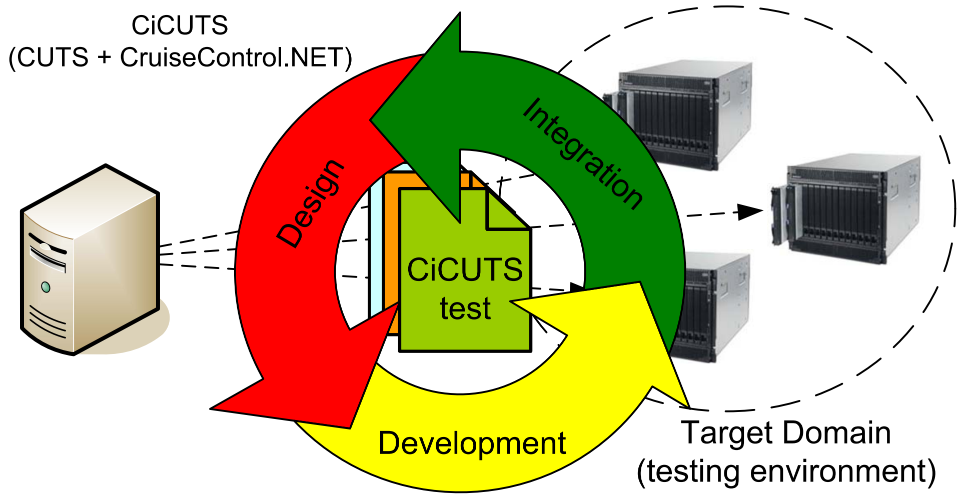


Figure VI.1: CiCUTS: Combining SEM tools with continuous integration environments

sure system QoS meets—or is close to meeting—its specification throughout the software lifecycle.

This chapter also describes how the methodology realized in CiCUTS can be applied to a case study of a representative large-scale component-based DRE system. The results from our case study show how combining continuous integration environments with DSML-based SEM tools enables system developers and testers to address the following requirements:

1. **Management and execution of large numbers of tests** by allowing system developers and testers to focus on systematic creation of test scenarios instead of expending time and effort building a custom testing framework. System developers and testers can create scenarios via CiCUTS's SEM tool and use its continuous integration environment to manage and execute them.

2. **Creation of realistic scenarios** by providing the capability of chaining scenarios that exercise multiple aspects of the DRE system and target environment's expected behavior. System testers can use CiCUTS to create tests and compose them into more complex/realistic tests scenarios that are managed and executed by its continuous integration environment.

3. **Automated processing and feedback of results** using graphical displays that visualize simple and detailed views of active and completed test scenarios. System developers and testers can use CiCUTS to ensure system performance is within QoS specification throughout the development lifecycle.

Chapter organization. The remainder of this chapter is organized as follows: Section VI.1 introduces our large-scale component-based DRE system case study; Section VI.2 describes how CiCUTS combines SEM tools with a continuous integration environment; Section VI.4 shows how we used CiCUTS to evaluate the QoS of our case study; and Section VI.5 summarizes the contributions of this chapter.

VI.1 Case Study: RACE and the Baseline Scenario

This section describes a large-scale component-based DRE case study from the domain of shipboard computing to motivate the need for, operation of, and benefits of integrating DSML-based SEM tools with continuous integration environments. This case study is based on the *Resource Allocation and Control Engine* (RACE) [111], which is an open-source distributed resource manager system we developed using the CIAO [28] implementation of the Lightweight CORBA Component Model (CCM) [89] over the past several years in conjunction with Lockheed Martin and Raytheon. RACE deploys and manages Lightweight CCM application component assemblies (henceforth called *operational strings*) based on specifications of their resource availability/usage and QoS requirements [111].

VI.1.1 Overview of RACE

Figure VI.2 shows the architecture of RACE, which is composed of four components assemblies (*Input Adapter*, *Planner Analyzer*, *Planner Manager*, and *Output Adapter*) that collaborate to manage operational strings for the target domain. RACE is designed to

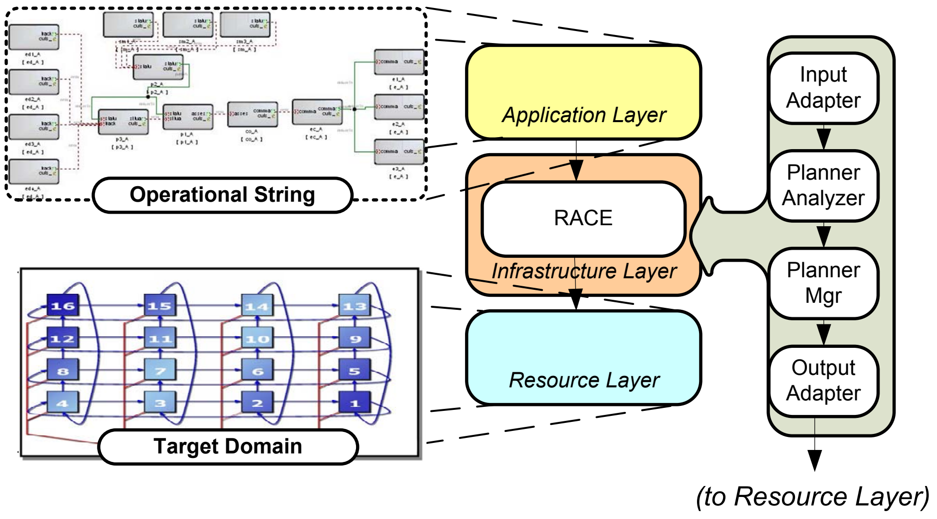


Figure VI.2: Architecture of RACE

perform two types of deployment strategies:

- **Static deployments** are operational strings created offline by humans or automated planners. RACE uses the information specified in a static deployment plan to map each component to its associated target host during the deployment phase of a DRE system. A benefit of RACE's static deployment strategy is its low runtime overhead since deployment decisions are made offline; a drawback is its lack of flexibility since deployment decisions cannot adapt to changes at runtime.

- **Dynamic deployments**, in contrast, are operational strings generated online by humans or automated planners. In dynamic deployments, components are not given a target host. Instead, the initial deployment plan contains component metadata (*e.g.*, connections, CPU utilization, and network bandwidth) that RACE uses to map components to associated target hosts during the runtime phase of a DRE system. A benefit of RACE's dynamic deployment strategy is its flexibility since deployment decisions can adapt to runtime changes (*e.g.*, variation in resource availability); a drawback is its higher runtime overhead.

VI.1.2 RACE's Baseline Scenario

The case study in this chapter focuses on RACE's *baseline scenario*. This scenario exercises RACE's ability to evaluate resource availability (*e.g.*, CPU utilization and network bandwidth) with respect to environmental changes (*e.g.*, node failure/recovery). Moreover, it evaluates RACE's ability to ensure that the lifetime of higher importance operational strings deployed dynamically is greater than or equal to the lifetime of lesser importance operational strings deployed statically based on resource availability.

Since RACE performs complex distributed resource management services—and thus took several years to develop—we wanted to avoid the serialized-phasing development problem outlined in Section I. In particular, we did not want to wait until final system integration to determine whether RACE could evaluate resource availability with respect to environmental changes to properly manage operational strings deployed dynamically versus those deployed statically.

To avoid the problems outlined above, we used CUTS and the techniques introduced in the previous chapters to analyze RACE's ability to manage the operational strings with respect to resource availability and environmental changes well before system integration to determine if we are meeting its QoS requirements. Moreover, to continuously ensure we are meeting the QoS requirements for RACE as we developed it, we combined CUTS with the CruiseControl.NET continuous integration environment to create CiCUTS. The remainder of this chapter describes CiCUTS and the results of our experiments that apply it to evaluate RACE's baseline scenario throughout its development lifecycle.

VI.2 Overview of CiCUTS

CiCUTS is a combination of continuous integration environments, such as CruiseControl.NET, and the CUTS SEM tool, which are two separately existing tools that work individually as follows:

- CruiseControl.NET monitors source code repositories for changes at predefined intervals. When changes are detected, it executes NAnt scripts that contain subtasks for performing work, such as building the application or executing unit tests. CruiseControl.NET then uses the return status of the NAnt scripts, which is based on the return value of the individual subtasks, to determine the success or failure of the entire process for the detected modification.

- CUTS uses profiling techniques to capture performance metrics of executing systems. It uses intrusive [77] and non-intrusive [72, 95] monitoring to capture metrics such as the service times of events in a component, the number of events received on each individual port of a component, or the execution time of a database query.

By combining CruiseControl.NET with CUTS, CiCUTS provides developers and testers with tools and analysis capabilities to improve testing features offered by system execution modeling tools. Developers and testers create *CiCUTS tests* (see Figure VI.1), which are NAnt subtasks that drive CUTS and the testing environment to create realistic scenarios (Requirement 2). CruiseControl.NET then continuously manages and executes CiCUTS tests throughout the development lifecycle (Requirement 1). Consequently, developers and testers can focus on resolving performance issues identified by CiCUTS instead of spending time and effort manually deploying and analyzing performance tests (Requirement 3).

VI.3 Evaluating Design Alternatives for CiCUTS

Successfully combining SEM tools like CUTS with a continuous integration environment like CruiseControl.NET requires developers and testers to agree upon the following profiling decisions: (1) what type of metrics to collect from the instrumented system being analyzed, (2) how to capture the performance metrics efficiently, and (3) how to present the metrics to a continuous integration environment so it can determine the testing result, *e.g.*, success or failure, of the system being analyzed. When developing CiCUTS we identified several ways to combine SEM tools with continuous integration environments. Below we logically evaluate the pros and cons of three design alternatives we considered.

VI.3.1 Alternative 1: Extend profiling infrastructure of SEM tools to capture domain-specific metrics.

Approach. SEM tools provide profiling infrastructures to collect predefined performance metrics, such as execution times of events/function calls or values of method arguments. As shown in Figure VI.3, it may be feasible to extend the profiling infrastructure, *i.e.*, the SEM data collector, to capture domain-specific metrics, such as the amount of time needed to deploy an operational string.

Evaluation. A benefit of this approach is that it simplifies development of a complete profiling framework. System developers and testers can leverage the SEM tool's existing infrastructure to collect and present domain-specific metrics to the continuous integration environment. Moreover, it simplifies deciding how to capture the metrics because the existing profiling infrastructure already has a predetermined method and format for collecting performance metrics. Developers and testers need only convert their target metrics into a format understood by the SEM profiling infrastructure.

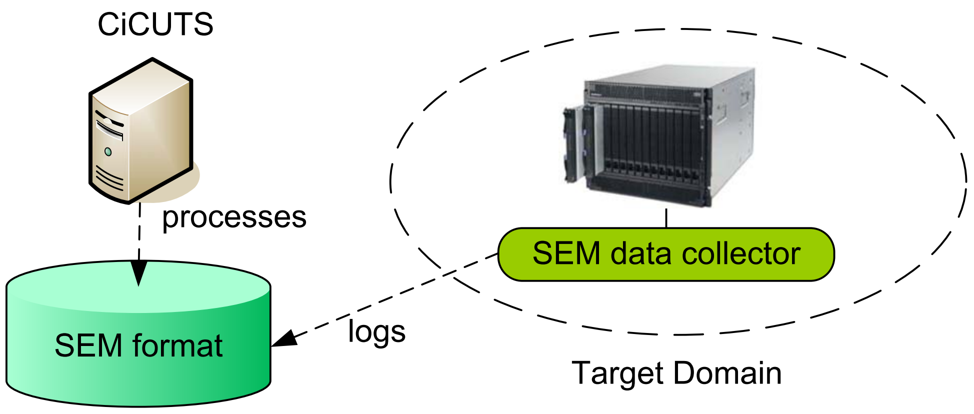


Figure VI.3: Conceptual model of design alternative 1

A drawback with this approach, however, is that it requires system developers and testers to ensure their domain-specific extensions to the SEM tool do not incur additional performance overhead on the instrumented system. For example, testers may collect metrics from complex data types, such as nested structures, that must be iterated in their entirety to obtain concrete data, but the runtime complexity of iteration process can adversely affect performance. Moreover, this approach may not be feasible if a SEM tool is proprietary, such that its profiling abilities cannot be extended by users.

VI.3.2 Alternative 2: Capture domain-specific performance metrics in format understood by continuous integration environments.

Approach. A continuous integration environment typically uses a predetermined format, such as verbose XML log files, to record and analyze the results of tests it manages. As shown in Figure VI.4, it may be feasible to capture target performance metrics outside of the profiling infrastructure using domain-specific data collectors and present performance metrics in the format understood by the continuous integration environment.

Evaluation. A benefit of this approach is that it simplifies integrating continuous integration environments with SEM tools because an understood format is used to present

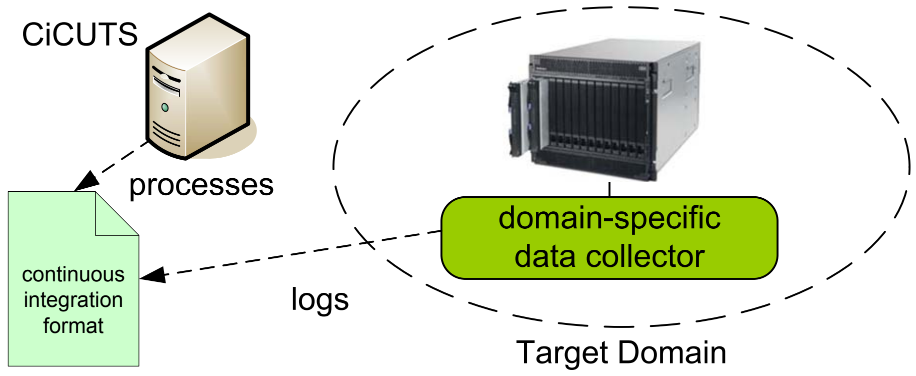


Figure VI.4: Conceptual model of design alternative 2

collected metrics. The continuous integration environment therefore already knows how to analyze the collected metrics and present the results.

A drawback with this approach, however, is that additional effort is required to develop a custom testing framework to collect performance metrics and feed them to the continuous integration environment. Moreover, this approach couples SEM tools with the continuous integration environment. If the project changes to a different continuous integration environment, then developers and testers must reimplement the testing framework to present metrics in a format understood by the new continuous integration environment.

VI.3.3 Alternative 3: Capture domain-specific performance metrics in an intermediate format.

Approach. SEM tools and continuous integration environments each have their own method and format for collecting and using performance metrics. As shown in Figure VI.5, it may be feasible to collect performance metrics outside of the existing profiling infrastructure—similar to alternative 2—but capture domain-specific metrics in a intermediate format that is not bound to any SEM tool or continuous integration environmentformat.

Evaluation. This approach applies the Bridge pattern [34] to capture metrics in an intermediate format, such as XML or a BLOB in a centralized database, that is neither

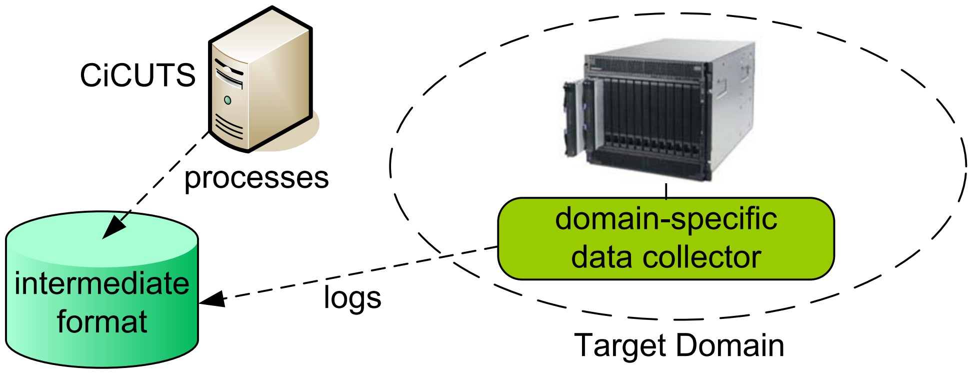


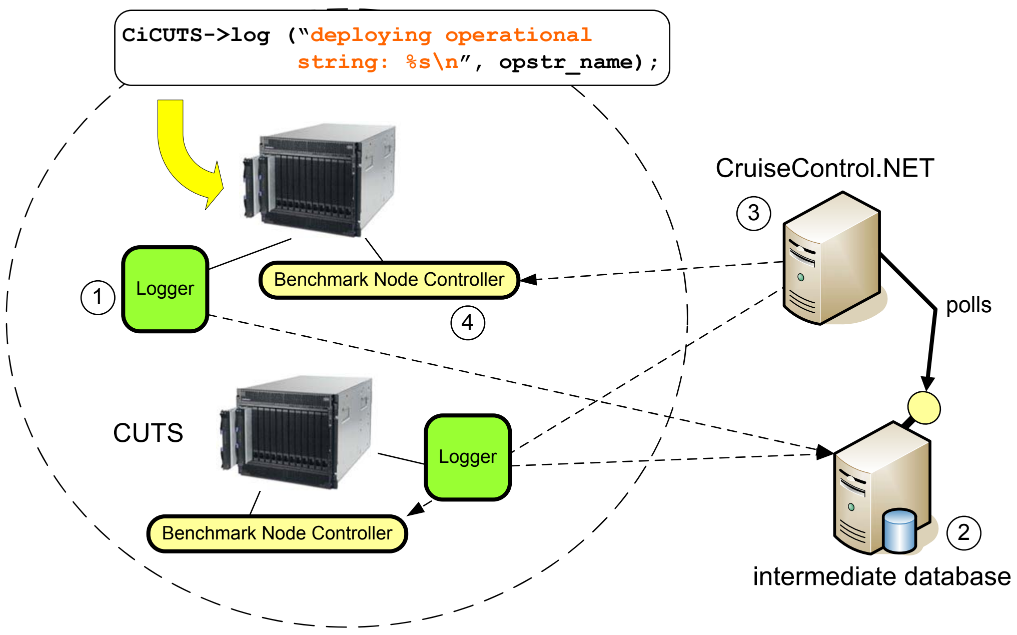
Figure VI.5: Conceptual model of design alternative 3

bound to a SEM tool nor the continuous integration environment format. A benefit of this approach is that it decouples the SEM tools from the continuous integration environment. Instead of presenting domain-specific performance metrics in a format understood by the continuous integration environment, developers and testers simply extend the continuous integration environment to understand the intermediate format for processing and analyzing results. Likewise, developers and testers can use any data collection technique, such as logging intercepters [50], to collect domain-specific performance metrics as long as they can transform the metrics into the intermediate format. Finally, the data collection technique does not interfere with the existing profiling infrastructure of the SEM tool.

A drawback with this approach, however, is that developers and testers must agree on the intermediate format to represent the data. Likewise, they must extend the continuous integration environment to understand the intermediate format for analyzing collected metrics. In practice, however, these drawbacks are not problematic because agreeing on an intermediate format is straightforward. Moreover, continuous integration environments are used in industrial development [14, 46] and support extension for domain-specific needs, such as evaluating the success of results generated by domain-specific extensions.

VI.3.4 The Structure and Functionality of CiCUTS

After evaluating the pros and cons of the various approaches described above, we selected alternative 3 for CiCUTS because it strongly decoupled of CUTS from CruiseControl.NET. As a result, if project collaborators decide to change to a different continuous integration environment they are not bound to using CruiseControl.NET. Likewise, if portions of RACE are redeveloped using a different SOA technology (*e.g.*, Microsoft.NET or J2EE), CruiseControl.NET can still be applied since it operates on the intermediate format, not the SEM tool's format. Moreover, testers and developers can use any data collection technique specific to the target SOA technology, such as the Java Messaging Service [119] for J2EE applications, as long as the collected performance metrics can be converted to the intermediate format and the data collection overhead is within an acceptable threshold. It is clear that alternative 3 offers the most flexibility when integrating CUTS with CruiseControl.NET to create CiCUTS.



```
CiCUTS->log ("deploying operational
             string: %s\n", opstr_name);
```

Figure VI.6: Structure of CiCUTS

Figure VI.6 shows the structure of CiCUTS, which is composed of the following elements: (1) *loggers*, which are domain-specific extensions to logging interceptors that transparently collect domain-specific performance metrics, (2) an intermediate *database* that stores performance metrics collected by the loggers, (3) *CruiseControl.NET*, which is CiCUTS's default continuous integration environment that manages and executes tests based on analyzed performance metrics, and (4) *Benchmark Node Controllers*, which execute commands directed by continuous integration environments, such as terminating container applications that host deployed operational strings. The loggers and the intermediate database in the CiCUTS infrastructure enable the combination of CUTS with CruiseControl.NET without tightly coupling one to the other.

To use CiCUTS, developers instrument their source code with log messages, *e.g.*, debug statements, that capture the desired performance metrics. We chose log messages because they do not limit which performance metrics can be collected, as long as the metrics can be represented as string literals. Likewise, testers create CiCUTS test scenarios using NAnt scripts that exercise different environment and system events, such as terminating/-recovering nodes that affect the lifetime of deployed operational strings (*i.e.*, Requirement 2). Finally, testers instruct CruiseControl.NET to manage and execute the CiCUTS tests (*i.e*, Requirement 1) by (1) monitoring the source code repository for modifications, (2) updating the testing environment with the latest development snapshot, (3) executing the CUTS tests scenarios, and (4) analyzing metrics in the intermediate database collected by the CUTS loggers (*i.e.*, Requirement 3).

VI.4 Continuous System Integration Testing Experiment & Results

This section presents the design and results of experiments that applied CiCUTS to evaluate the QoS of RACE's *baseline scenario* described in Section VI.1.2. These experiments evaluated the following hypotheses: (H1) CiCUTS allows developers to understand the behavior and performance of infrastructure-level applications, such as RACE, before system integration and (H2) CiCUTS allows developers to ensure that the QoS performance of infrastructure-level applications is within performance specifications throughout the development lifecycle more efficiently and effectively than waiting until system integration to evaluate performance.

VI.4.1 Experiment Design

To evaluate the two hypotheses in the context of the RACE baseline scenario, we constructed 10 operational strings. Each string was composed of the same components and port connections, but had different importance values and resource requirements to reflect varying resource requirements and functional importance between operational strings that accomplish similar tasks, such as a primary and secondary tracking operation. Figure VI.7 shows a structural model for one of the baseline scenario's operational strings—which was

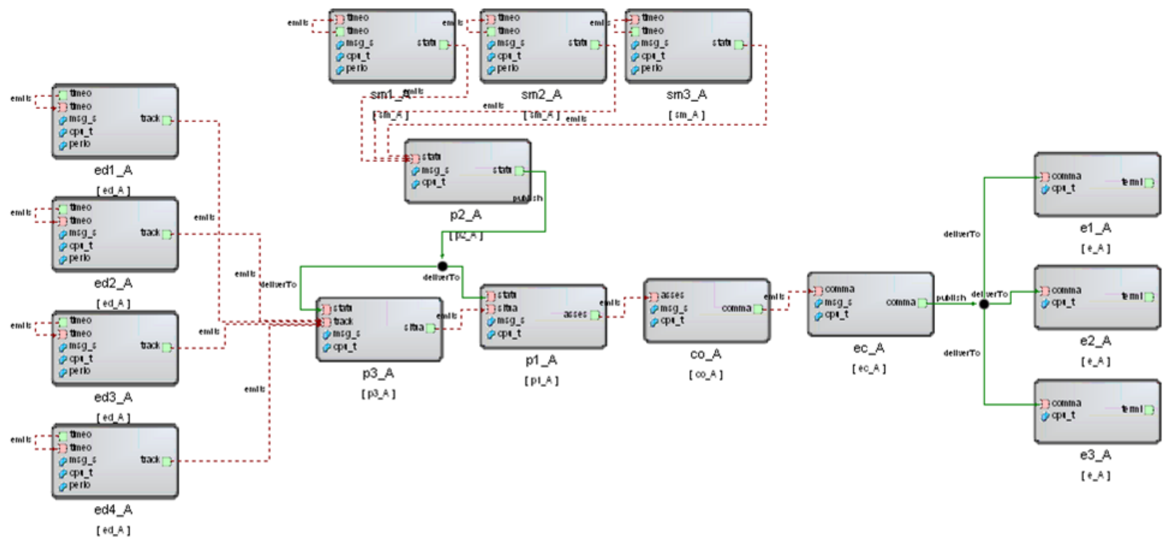


Figure VI.7: Structural model of the replicated operational string for the RACE baseline scenario

replicated 10 times to create the 10 operational strings in the baseline scenario—consisting of 15 interconnected components represented by the rounded boxes.

The four components on the left side of the operational string in Figure VI.7 are sensor components that monitor environment activities, such as tracking objects of importance using a radar. The four components in the top-middle of Figure VI.7 are system observation components that monitor the state of the system. The four linear components in the bottom-center of Figure VI.7 are planner components that receive information from both the system observation and sensor components and analyze the data, *e.g.*, determine if the object(s) detected by the sensor components are of importance and how to (re)configure the system to react to the detected object(s). The planner components then send their analysis results to the three components on the right side of Figure VI.7, which are effector components that react as stated by the planner components (*e.g.*, start recording observed data).

To prepare RACE's baseline scenario for CiCUTS usage (see Section VI.3.4), we used CUTS modeling languages, which are presented in Chapter III, to construct the 10 operational strings described above. We then used the generative techniques presented in Chapter IV to generate Lightweight CCM compliant emulation code that represented each component in the operational string managed by RACE (see Figure VI.7) in the baseline scenario. We also used PICML to generate the operational strings' deployment and configuration descriptors for RACE. The deployment for each string used the strategy is specified in Table VI.1. The importance values[1] assigned to each operational string reflects its mission-critical ranking with respect to other operational strings. We chose extreme importance values because RACE was in its initial stages of development and we wanted

Table VI.1: Importance values of the RACE baseline scenario operational strings

Operational String	Importance Value
A – H	90
I – J	2

[1]These values are not OS priorities; instead, they are values that specify the significance of operational strings to each other.

to ensure that it honored importance values when managing operational strings. Finally, we annotated RACE's source code with the logging mechanisms described in Section VI.2 to collect information, such as time of operational string deployment/teardown or time of node failure recognition.

To run the experiments using CiCUTS, we created NAnt scripts that captured the serialized flow of each experiment. The NAnt scripts contained commands that (1) signaled RACE to deploy/teardown operational strings, (2) sent commands to individual nodes to cause environmental changes, and (3) queried the logging database for test results. The CruiseControl.NET part of CiCUTS then used the NAnt scripts to manage and execute the experiments many times, *e.g.*, every 30 minutes CruiseControl.NET checked for modifications in the RACE source code repository and, if so, executed the NAnt scripts.

When the RACE baseline scenario tests are executed under control of CruiseControl.NET, log messages containing the information outlined above were generated when the RACE's runtime execution reached that point of execution. These log messages were stored in a database by the CUTS logger's in CiCUTS for offline analysis by CruiseControl.NET, *e.g.*, calculating the lifetime of operational strings or amount to time to deploy operational strings and representing it as an integer value. The collected log messages were also transformed into a graphically display (*e.g.*, see Figure VI.8) to show whether the lifetime of dynamic deployments exceed the lifetime of static deployments based on resource availability with respect to environmental changes.

VI.4.2 Experiment Results

This section presents the results of experiments that validate H1 and H2 about CiCUTS when evaluating the QoS of the RACE baseline scenario.

VI.4.2.1 Using CiCUTS to Understand the Behavior and Performance of Infrastructure-level Applications

H1 conjectures that CiCUTS will assist in understanding the behavior and performance of infrastructure-level applications, such as RACE, well before system integration. Figure VI.8 shows an example result set for the RACE baseline scenario (*i.e.*, measuring the lifetime of operational strings deployed dynamically vs. operational strings deployed statically) where 2 hosts were taken offline to simulate a node failure. The graphs in Fig-

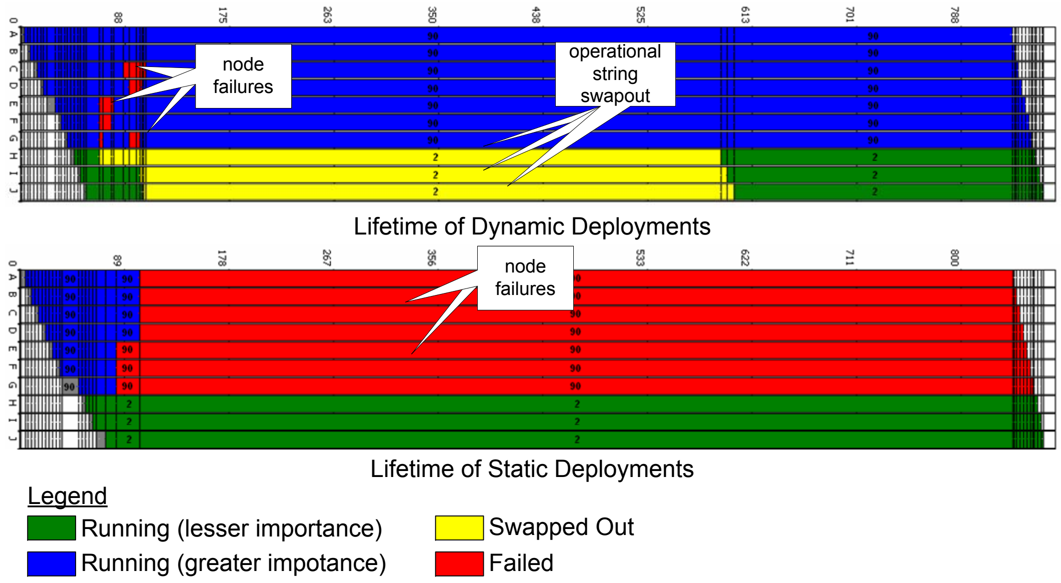


Figure VI.8: Graphical analysis of static deployments (bottom) vs. dynamic deployments (top) using RACE

ure VI.8—which are specific to RACE—were generated from the log messages stored in the database via the CUTS loggers described in Section VI.3. The x-axis in both graphs is the timeline for the test in seconds and each horizontal bar represents the lifetime of an operational string, *i.e.*, operational string A-J.

The graph at the bottom of Figure VI.8 depicts RACE's behavior when deploying and managing human-generated static deployment of operational string A-J. The graph at the top of Figure VI.8 depicts RACE's behavior when deploying and managing RACE-generated dynamic deployment of operational string A-J. At approximately 100 and 130 seconds into the test run we instructed the Benchmark Node Controller to randomly kill 2 nodes hosting the higher importance operational strings, which is highlighted by the "node failures" callout.

As shown in the static deployment (bottom graph) of Figure VI.8, static deployments are not aware of the environmental changes. All operational strings on failed nodes (*i.e.*, operational string A-G) therefore remain in the failed state until they are manually redeployed. In this test run, however, we did not redeploy the operational strings hosted on the failed nodes because the random "think time" required to manually create a deployment and configuration for the 7 failed operational strings exceeded the duration of the test. This result signified that in some cases it is too hard to derive new deployments due to stringent resource requirements and scarce resource availability.

The behavior of dynamic deployment (top graph) is different than the static deployment (bottom graph) behavior. In particular, when the Benchmark Node Controller kills the same nodes at approximately the same time (*i.e.*, section highlighted by the "node failure" callout), RACE's monitoring agents detect the environmental changes. RACE then quickly tears down the lower importance operational strings (*i.e.*, the section highlighted by the "operational string swapout") and redeploys the higher importance operational strings in their place (*e.g.*, the regions after the "node failure" regions).

The test run shown in Figure VI.8, however, does not recover the failed nodes to emulate the condition where the nodes cannot be recovered (*e.g.*, due to faulty hardware). This failure prevented RACE from redeploying the lower importance operational strings because there were not enough resources available. Moreover, RACE must ensure the lifetime of the higher importance operational strings is greater than lower importance operational strings (see Section VI.1). If the failed nodes were recovered, however, RACE would attempt to redeploy the lower importance operational strings. Figure VI.8 also shows the lifetime of higher importance operational strings was $\sim$15% greater than lower importance operational string. This test case showed that RACE can improve the lifetime of operational strings deployed and managed dynamically vs. statically.

The results described above validate H1, *i.e.*, that CiCUTS enables developer to understand the behavior and performance of infrastructure-level applications. Without CiCUTS, we would have used *ad hoc* techniques, such as manually inspecting execution trace logs distributed across multiple hosts, to determine the exact behavior of RACE. By using CiCUTS, however, we collected the necessary *log messages* in a central location and used them to determine the exact behavior of RACE. Moreover, the collected log messages helped determine if RACE was performing close to its QoS specifications. Without CiCUTS, not only would we have had to rely on *ad hoc* techniques to understand the behavior of RACE and evaluate its performance, we would not have been able to do so well in advance of final system integration.

VI.4.2.2 Using CiCUTS to Ensure Performance is Within QoS Specifications

H2 conjectures that CiCUTS would help developers ensure the QoS of infrastructure-level applications is within its performance specifications throughout the development lifecycle. The results described in Section VI.4.2.1, however, represent a single test run of the baseline experiment. Although this result is promising, it does not show conclusively that CiCUTS can ensure RACE is within its QoS specifications as we develop and release revisions of RACE.

We therefore used the CruiseControl.NET portion of CiCUTS to continuously execute variations of the experiment previously discussed while we evolved RACE. Figure VI.9 highlights the maximum number of tests we captured from the baseline scenario presented in Figure VI.8 after it was executed approximately 427 times over a 2 week period. The number of executions corresponds to the number of times a modification (such as a bug

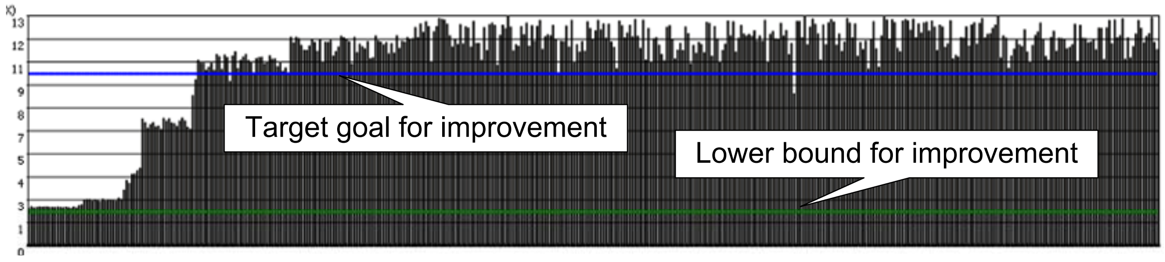


Figure VI.9: Overview analysis of continuously executing the RACE baseline scenario

fix or an added feature to RACE) was detected in the source code repository at 30 minute intervals.

The vertical bars in Figure VI.9 represent the factor of improvement of dynamic deployments vs. static deployments. The heights of the bars in this figure are low on the left side and high on the right side, which stem from the fact that the initial development stages of RACE had limited capability to handle dynamic (re-)configuration of operational strings. As RACE's implementation improved—and the modified code was committed to the RACE source code repository—the CruiseControl.NET portion of CiCUTS updated the testing environment automatically. The results in Figure VI.9 show how the CruiseControl.NET part of CiCUTS manages and executes tests of RACE's baseline scenario efficiently because it automatically monitors the source code repository and reruns the tests if modifications are detected.

The results in Figure VI.9 also show how CiCUTS allows developers to keep track of RACE's performance throughout its development. As the performance of RACE improved between source code modifications, the vertical bars increased in height. Likewise, as the performance of RACE decreased between source code modifications, the vertical bars decreased in height. Lastly, since each vertical bar corresponds to a single test run, if the performance of RACE changed between tests runs, developers could look at the graphical display for a single test run (see Figure VI.8) to further investigate RACE's behavior.

The results described above validate H2, *i.e.*, that CiCUTS helps developers ensure the

QoS of infrastructure-level applications is within its performance specifications throughout the development lifecycle. As modifications where checked into the source code repository, the CruiseControl.NET portion of CiCUTS detected the modifications and reran the QoS tests. As shown in Figure VI.9, each set of modifications within a predefined time period (*e.g.*, every 30 minutes) corresponded to a single test run. As performance improved or declined, developers can locate which modifications resulted in the changes.

Conducting this process without CiCUTS is hard because it requires testers to manually (1) monitor the source code repository, (2) update the testing environment, (3) rerun the performance tests, and (4) associate the test results with detected modifications. In contrast, CiCUTS helps ensure system performance is within is QoS specifications more efficiently and effectively. Its key contribution is automating the testing process and autonomously providing feedback about whether the system is or is not within its QoS specification.

VI.5 Chapter Summary

This chapter described the design and application of CiCUTS, which combines the DSML-based SEM tools, such as CUTS, with continuous integration environments. We evaluated the design alternatives we considered when integrating CUTS with continuous integration environments and explained the structure and functionality of the approach we selected. We also presented a case study that applied CiCUTS to a representative large-scale component-based DRE system—called RACE—to evaluate its QoS and perform integration testing continuously throughout its development process to validate how well revisions to the RACE software met—or did not meet—their QoS requirements.

Our case study showed how CUTS leveraged continuous integration capabilities to enhance its testing capabilities. More importantly, the integration of CUTS with continuous integration environments bridges the gap between agile development methodologies and QoS evaluation for large-scale component-based DRE systems plagued by serialized-phasing development.

CHAPTER VII

TEMPLATE PATTERNS FOR IMPROVING CONFIGURABILITY AND SCALABILITY OF TESTING AND EXPERIMENTATION

Large-scale component-based DRE systems, *e.g.*, shipboard computing environments, mission avionic systems, and air traffic control, possess characteristics that complicate verification and validation [51], such as complexity, heterogeneity, and scale. Testing and experimentation (T&E), *i.e.*, running many tests of the system under different configurations to exercise validation of large-scale component-based DRE systems, is the conventional method for evaluating system behavior and quality-of-service (QoS). T&E of large-scale component-based DRE systems in realistic environments and operating conditions also helps increase confidence that the system being developed meets its functional and QoS requirements [22, 47]. For example, large-scale component-based DRE system developers can leverage dynamically configurable testbeds, such as Emulab [101], to produce realistic environments for understanding and evaluating system QoS throughout the software lifecycle.

The T&E process, however, can be expensive and time consuming [47], even when it is automated. Moreover, the effectiveness of T&E relates directly to the ability to test many different system configurations (such as the number of hosts, the hreading model, and the number of clients) in realistic and hypothetical operating conditions. For example, understanding and evaluating the worst-case response time of critical execution paths in a DRE system requires testing a DRE system under different workloads.

Existing techniques that address T&E configuration concerns include domain-specific modeling languages [65], which can help alleviate the complexity of handcrafting configuration files via models and model interpreters. These techniques, however, rely largely on *single instance* configurations, where one configuration is used to evaluate a system under a single operating condition. To evaluate the system under different operating conditions, large-scale component-based DRE system developers must produce multiple variants of the same configuration file (or model), which is tedious, time-consuming, and error-prone. Developers therefore need improved techniques and patterns to improve T&E configurability and scalabilty to broaden their evaluation scope.

Solution approach: Template patterns. A *template* is an abstraction that captures the fixed and variable portions of a context, such as an algorithm [3, 24], model transformation [56, 117], and configuration file [37, 125]. For example, the Template Method [34] design pattern enables developers to capture the fixed portion of an algorithm while deferring certain variable steps to subclasses. *Template patterns* describe techniques for transforming the variable portion of a given template, such as setting a variable in a configuration file's template based on it hostname. In general, templates help increase the configurability and scalability of their application context, including offline situations where timing constraints are not an issue.

This chapter describes the following related template patterns that help increase T&E configurability and scalability: *Variable Configuration*, *Batch Variable Configuration*, *Dynamic Variable Configuration*, and *Batch Dynamic Variable Configuration*. We have implemented variants of each pattern in the *Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS) Template Engine* (CUTE). CUTE uses a domain-specific language to capture fixed and variable portions of T&E configurations, such as resolving the correct network interface based on the operating environment of a test. Large-scale component-based DRE system developers can use CUTE to decrease the number of *single instance* configuration files for T&E, while increasing T&E configurablity and scope. This chapter shows how the template patterns supported by CUTE help improve large-scale component-based DRE systems evaluation capabilities, such as testing hypothetical workloads or migrating between different operating environments.

Chapter organization. The remainder of this chapter is organized as follows: Section VII.1 revisits an existing case study of a representative large-scale component-based DRE system project to motivate the need for CUTE; Section VII.2 describes the four template patterns that CUTE implements to improve T&E configurability and scalabilty; Section VII.3 provides quantitative analysis of these template patterns in the context of the case study; and Section VII.4 summarizes the contributions of this chapter.

VII.1 Case Study: Revisiting the QED Project

In Chapter V Section V.1, we introduced the the QoS-Enabled Dissemination (QED) [1] project, which is a large-scale, multi-team collaborative project involving Vanderbilt, Boeing, BBN Technologies, and IHMC aimed at addressing QoS concerns within the Global Information Grid (GIG) [2]. As explained in Chapter V Section V.1, the goal of the QED project is to improve QoS concerns of the GIG, which involves evaluating many points-of-variability, such as scalablity, operating environments, and workload. To evaluate that GIG QoS concerns are addressed adequately, QED developers plan to test many different candidate technologies and implementations, including Mockets [126] and differentiated service queues [6, 52] under various deployment scenarios.

Conventional T&E techniques, such as handcrafted and hardcoded scripts or conventional DSMLs, require QED testers to expend much time and effort conducting T&E on the QED middleware. For example, if QED testers used domain-specific modeling languages to auto-generate configuration files, they would have to create a model for each single instance configuration. More specifically, QED testers are faced with the following challenges:

- **Varying T&E scenario configurations with minimal effort.** Each T&E scenario in the QED project is expensive in both time and effort to create. Forcing QED testers to (re)implement each scenario for each different configuration is also an expensive—yet tedious and error prone—task to undertake. QED testers, therefore, need techniques that will enable them to define a T&E scenario once and vary its configuration without incurring the effort required to realize the original T&E scenario.

- **Supporting multiple testers and operational environments for single T&E scenarios.** Each stackholder in QED, such as BBN Technologies, Boeing, IHMC, and Vanderbilt, have testers responsible for evaluating their respective development features in QED against the GIG middleware under different scenarios. Moreover, all tests are conducted in a dynamic testing environment named ISISlab (see Appendix A). Forcing QED testers to manage duplicate versions of a single T&E scenario to account for the dynamics and heterogeneity of each individual tester and the target testing environment is an expensive process. QED testers therefore need better techniques to reduce such complexity when defining T&E scenarios.

The remainder of this chapter discusses four T&E template patterns we identified, implemented, and analyzed to address the challenges of QED testers and to simplify T&E efforts, while increasing evaluation capabilities of QED.

VII.2 Template Patterns for Testing and Experimentation

This section discusses four template patterns for T&E that increase configurability and scalability. As shown in Figure VII.1, each pattern builds upon the other, and are the building blocks of CUTE. The remainder of this chapter provides a detailed synopsis (*i.e.*, problem statement and solution), evaluation, and application of each pattern in the context of CUTE and the QED project case study described in Section VII.1.

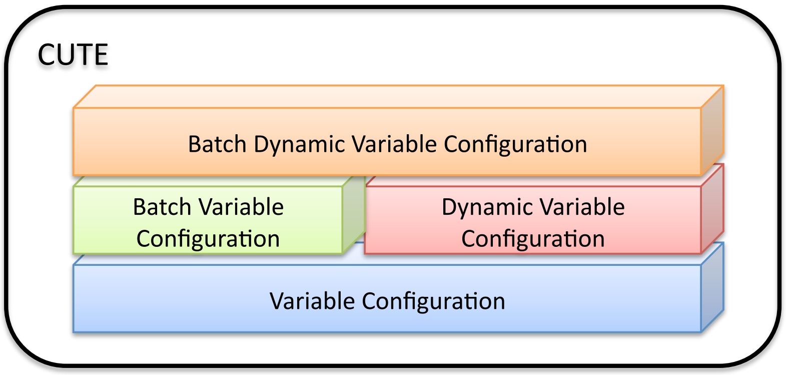

CUTE
Batch Dynamic Variable Configuration
Batch Variable Configuration
Dynamic Variable Configuration
Variable Configuration

Figure VII.1: Template patterns that are building blocks for CUTE

VII.2.1 Variable Configuration Pattern

Problem statement. Traditional techniques for T&E rely on configuration files to determine the test scenario for the large-scale component-based DRE system under development. The benefit of configuration files is that they decouple the implementation from the configuration and behavior, *i.e.*, enable late-binding. For example, component-based middleware [90, 93] uses verbose XML files that determine how components intercommunicate, what properties to set for the underlying middleware, and what values to set for attributes of a component. System developers therefore implement the static portion of the large-scale component-based DRE system in terms of the variable portion, and let the variable portion (which realizes the decoupling) be controlled by external configuration files.

In T&E, a single instance configuration file represents a single instance of a test run. To evaluate a large-scale component-based DRE system, such as QED, under many different scenarios (*i.e.*, configurations), system developers must manually produce many different configurations. In many cases, however, the size of the static portion of the configuration has greater cost (*e.g.*, time to generate, or number of characters) than the variable portion of the configuration. Even in cases when the cost of the static portion is less than the cost of the variable portion, manually producing multiple variants is both time-consuming and error prone.

Solution. The *Variable Configuration* pattern allows large-scale component-based DRE system developers to define the static portion of a configuration file, such as the required header information in an XML document, while using variables (or placeholders) to capture the variable portion of the configuration. We formally define the Variable Configuration pattern $C = (S, V)$ as:

- a set S of characters that capture the static portion of the configuration C, and
- a set V of variables that define the variable portion of the configuration C.

A *configuration instance* is a single configuration used to execute a test scenario. It is derived from a template configuration C by replacing the variable portions V of the configuration with concrete values. Equation VII.1 defines the equation used to evaluate a template configuration C.

$$C' = eval(C, D) \tag{VII.1}$$

where C' is a single instance configuration of C, and D is a set of tuples (K, v) determined by system testers such that $K \in V$ and $v = value(K)$ for configuration C'—which is analogous to a dictionary.

Manifestation in CUTE and QED. We have realized the Variable Configuration pattern in CUTE and applied it to the QED project. QED testers utilize the Variable Configuration pattern using the following steps:

1. Define a text-based file that captures a single configuration.
2. Replace static portions of the single configuration with a template variable. This represents a point-of-variablity in the template configuration.[1]
3. Define a dictionary file that consists of all key-value pairs for each variable in the Variable Configuration. Developers also have the option of overriding keys in the dictionary file at the command-line when invoking CUTE.

[1]It is possible to combine this step and the previous step by auto-generating a template with its variables already defined.

Using the user-defined template and dictionary, CUTE applies Equation VII.1 to produce a single instance configuration. QED testers then use the derived configuration to evaluate the system under development.

```
 1  ...
 2    <configProperty>
 3      <name>cpuTime</name>
 4      <value>
 5        <type><kind>tk_long</kind></type>
 6        <value><long>${cpuTime}</long></value>
 7      </value>
 8    </configProperty>
 9    <configProperty>
10      <name>testOwner</name>
11      <value>
12        <type><kind>tk_string</kind></type>
13        <value><string>${testOwner}</string></value>
14      </value>
15    </configProperty>
16  ...
```

Listing VII.1: Example of the deployment and configuration that uses the Variable Configuration pattern

Listing VII.1 shows an excerpt from the deployment and configuration (D&C) file of a test scenario from the QED project. Listing VII.1 contains two variables named `cpuTime` (line 6) and `testOwner` (line 13), which are points-of-variability. Likewise, Listing VII.2 is an example dictionary for the template configuration in Listing VII.1, and Listing VII.3 shows the concrete instantiation of the template configuration using the dictionary in Listing VII.2.

```
cpuTime=33.4
testOwner=hillj
```

Listing VII.2: Dictionary for the D&C template.

```
...
  <configProperty>
    <name>cpuTime</name>
    <value>
```

```
        <type><kind>tk_long</kind></type>
        <value><long>33.4</long></value>
      </value>
    </configProperty>
    <configProperty>
      <name>testOwner</name>
      <value>
        <type><kind>tk_string</kind></type>
        <value><string>hillj</string></value>
      </value>
    </configProperty>
...
```

Listing VII.3: Evaluation of the Variable Configuration pattern for the D&C.

Without the Variable Configuration pattern of the configuration, QED testers would have to define each single instance configuration manually. Moreover, CUTE helps increase the configurability and flexibility of T&E configuration by deferring realization of a single instance configuration until as late as possible, *i.e.*, not solely at test design time.

Pattern evaluation. The following is a list of benefits for using the Variable Configuration pattern:

- It reduces the number of single instance configurations that must be handcrafted since system testers have to produce a template only once and use a dictionary to define each configuration instance.

- It reduces the amount of error that can be incurred from manually replicating each configuration and modifying a small portion of it. In the case of tools that can auto-generate configuration files, *e.g.*, domain-specific modeling languages, it reduces the amount of effort required to produce each configuration, such as manually modeling each configuration via a cumbersome and tedious process. This also helps increase the scalability of T&E.

- It highlights the points-of-variability in a configuration. This makes is easier for large-scale component-based DRE system developers to know what are the control parameters in a test scenario (or configuration)—similar to control parameters in an experiment. This also helps increase the scope of T&E.

Although the Variable Configuration pattern has many benefits, it also has several consequences. The following is a list of consequences for using the Variable Configuration pattern:

- Developers must define the value of each variable in V, or the configuration is considered to be *invalid* since all variables do not expand to a concrete value. This can become problematic in situations where portions of the Variable Configuration can be ignored, such as doing multiple passes in a single template using different configuration dictionaries and each configuration dictionary contains a disjoint subset of the variables in the template configuration. Developers must take this scenario into account to ensure each variable is properly expanded before using the single instance configuration.

- The *usability* of a configuration is unknown until it is applied to the system under development in a test scenario. This can be overcome, however, if the mechanism for generating the dictionary understands the contraints of the target context, similar to constraints in domain-specific modeling languages.

VII.2.2 Batch Variable Configuration Pattern

Problem statement. In Section VII.2.1, we discussed the Variable Configuration pattern and how it enables developers to generate single instance configuration files using a template and dictionary. This technique is acceptable when testing only a few different configurations since the number of dictionaries required to realize each single instance configuration is minimal. For example, if a developer wants to validate a single configuration by executing a test run of the configuration, then the Variable Configuration pattern is a satisfactory technique.

When trying to test many different configurations derived from a single template, however, it becomes problematic trying to manage many different dictionary files—where each instantiates a single instance configuration. Moreover, it becomes hard to logically understand how each dictionary relates without using *ad hoc* techniques, such as creating test suites where each configuration is representative of each test in the test suite. For example, a QED tester who is testing performance and security logically maintains each dictionary for evaluating the respective concern in a directory that has the appropriate name, such as `./performacnce` or `./security`. Although this is acceptable, it negatively impacts T&E maintainability and scalability.

Solution. The *Batch Variable Configuration* pattern builds upon the Variable Configuration pattern and enables large-scale component-based DRE system developers to logically group common configurations (*i.e.*, dictionaries) that are evaluated at once. Developers leverage the Batch Variable Configuration pattern by defining logically related dictionaries, such that all the dictionaries used to generate configurations for evaluating performance, in a single monolithic configuration.

The Batch Variable Configuration has the same formal definition as the Variable Configuration (see Section VII.2.1) since it is using a single template. It, however, has a different evaluation function as illustrated by Equation VII.2:

$$C'' = batcheval(C, D') = \{\forall d \in D' : eval(C, D)\} \tag{VII.2}$$

where D' is the set of dictionaries such that if $d \in D'$ then d is an instance of D (see Section VII.2.1) and C'' is the set of configurations such that $C' = eval(C, d)$ and $C' \in C''$.

Manifestation in CUTE and QED. We have realized the Batch Variable Configuration pattern in CUTE and applied it to the QED project. QED testers leverage the Batch Variable Configuration pattern using the following steps:

1. Define a text-based template configuration file that contains variables, similar to the process in the Variable Configuration pattern (see Section VII.2.1).

2. Define a set of dictionaries in a single file where each individual dictionary defines the key-value pair for each variable in the template configuration file.

Using the user-defined template file and batch configuration file, CUTE applies Equation VII.2 to produce a set of configuration files. QED testers then use the derived configuration files to evaluate system QoS.

```
1 config (lowCPU.cdp) {
2   cpuTime=33.4
3   testOwner=hillj
4 }
5
6 config (highCPU.cdp) {
7   cpuTime=87.8
8   testOwner=hillj
9 }
```

Listing VII.4: Batch dictionary for the D&C template.

Listing VII.4 illustrates a set of dictionaries for the template configuration in Listing VII.1. As highlighted in Listing VII.4, there are two different, yet related, configurations for testing CPU workload named: `lowCPU.cdp` (line 1) and `highCPU.cdp` (line 6). Likewise, Listing VII.5 shows the concrete instantiation of the D&C excerpt for the batch configurations in Listing VII.4.

```
// lowCPU.cdp
...
  <configProperty>
    <name>cpuTime</name>
    <value>
      <type><kind>tk_long</kind></type>
      <value><long>33.4</long></value>
    </value>
  </configProperty>
  <configProperty>
```

```
      <name>testOwner </name>
      <value>
         <type><kind>tk_string </kind></type>
         <value><string>hillj </string></value>
      </value>
   </configProperty>
...

// highCPU.cdp
...
  <configProperty>
      <name>cpuTime </name>
      <value>
         <type><kind>tk_long </kind></type>
         <value><long>87.8 </long></value>
      </value>
   </configProperty>
  <configProperty>
      <name>testOwner </name>
      <value>
         <type><kind>tk_string </kind></type>
         <value><string>hillj </string></value>
      </value>
   </configProperty>
...
```

Listing VII.5: Evaluation of the Batch Variable Configuration pattern for the D&C.

Without the Batch Variable Configuraton pattern, QED testers would have a hard time logically maintaining and producing multiple configuration files used to evaluate system QoS.

Pattern evaluation. In addition to the benefits for using the Variable Configuration pattern, the following is a list of benefits of using the Batch Variable Configuration pattern:

- It reduces the amount of single configurations that must be managed by large-scale component-based DRE system developers. Developers can just define all *valid* configurations in a single file and batch process it.

- It helps logically group together different configurations that can be processed at once. For example, using the Batch Variable Configuration pattern, developers can group configurations by ownership (*i.e.*, who created the configuration) or by QoS concerns for each individual configuration tests, such as performance, reliability, and security.

Although the Batch Variable Configuration patterns has many benefits, it also has several consequences. In addition to the consequences of the Variable Configuration pattern, the following is a list of consequences for using the Batch Variable Configuration pattern:

- The Batch Variable Configuration is *valid* if and only if each individual configuration is valid. It is, however, possible to suppress this consequence through *partial validity* where it is acceptable to have invalid configurations in the batch configuration, or ensuring each individual configuration is valid before including it in the batch configuration.

- The Batch Variable Configuration can be a *point-of-failure* because multiple configurations are defined in a single file. If the batch file is lost, then testers must redefine the configuration file. This can be overcome, however, by chaining a batch configuration using external configuration—similar to the C++ include preprocessor definition.

VII.2.3 Dynamic Variable Configuration Pattern

Problem statement. In Section VII.2.1 and Section VII.2.2, we discussed two template patterns for improving both configurability and scalability of T&E. Both patterns, however, do not fully take into account T&E concerns, such as the operating environment. For example, `testOwner` in Listing VII.2 and Listing VII.4 is hardcoded to `hillj`. There can be situations where the value of `testOwner` needs to be determined by the username of the person evaluating the configuration for accountability purposes.

Requiring large-scale component-based DRE system developers to maintain many different configuration files based on some side-effect of the operating environment—even in the case of the Batch Variable Configuration pattern—can negatively affect adaptability and configurability. For example, if developers are using dynamic testing environments, such as ISISlab, and want to evaluate performance of the same large-scale component-based DRE system under different experiments, then it requires developers to maintain different (yet similar) configurations for each experiment since the operating environment is *logically* different, *e.g.*, different hostnames and IP addresses.

Solution. The *Dynamic Variable Configuration* pattern builds upon the Variable Configuration pattern and enables developers to capture the variable portion of a configuration that depends on a context-based side-effect, such as the hostname of a node in an experiment or the output of an equation evaluator. We formally define the Dynamic Variable Configuration pattern $DC = (C, \Delta)$ as:

- a template configuration C that contains the static and variable portions of the configuration file (see Section VII.2.1), and

- a set Δ of context-based side-effects (or dynamic variables) that capture dynamic portions of configuration DC. We assume that $\delta \in \Delta$ produces a single line of text that can be used in place of its respective dynamic variable in C.

We derive a single instance configuration of DC by first evaluating C using Equation VII.1 in Section VII.2.1. Next, we evaluate the result C' using Equation VII.3:

$$DC' = eval(C', \Delta) \tag{VII.3}$$

where DC' is a single instance configuration for the Dynamic Variable Configuration pattern that large-scale component-based DRE system developers use for T&E.

Manifestation in CUTE and QED. We have realized the Dynamic Variable Configuration pattern in CUTE and applied it to the QED project. QED testers leverage the Dynamic Variable Configuration pattern using the following steps:

1. Define a text-based template configuration file that contains variables similar to the process in the Variable Configuration pattern (see Section VII.2.1).

2. Replace portions of the configuration with dynamic variables that will perform a side-effect to determine the value for that particular portion of the document.

3. Define a dictionary file that consists of all key-value pairs for each variable in the configuration. Developers also have the option of overriding keys in the dictionary file at the command-line when invoking CUTE.

Using the user-defined template, dictionary, and side-effects, CUTE uses Equation VII.1 and Equation VII.3 to evaluate the configuration. QED testers then use the derived single instance configuration to evaluate system QoS.

```
 1  ...
 2    <configProperty>
 3      <name>cpuTime</name>
 4      <value>
 5        <type><kind>tk_long</kind></type>
 6        <value><long>${cpuTime}</long></value>
 7       </value>
 8    </configProperty>
 9    <configProperty>
10      <name>testOwner</name>
11      <value>
12        <type><kind>tk_string</kind></type>
13        <value><string>${userName}</string></value>
14       </value>
15    </configProperty>
16  ...
```

Listing VII.6: Example of the D&C that uses the Dynamic Variable Configuration pattern

Listing VII.6 illustrates an example configuration that uses the Dynamic Variable Configuration pattern, which is a variant of the configuration from Listing VII.1. As highlighted on line 13, we have replaced `hillj` with a variable named `userName`. Likewise, Listing VII.7 lists an example dictionary for Listing VII.6, which produces the same single instance configuration in Listing VII.3[2]. In this case, `userName` is defined as a dynamic variable where `userName` is determined by the output of the command `/usr/bin/whoami`.

```
cpuTime=33.4
userName=`/usr/bin/whoami`
```

Listing VII.7: Example dictionary for the Dynamic Variable Configuration pattern.

Without the Dynamic Variable Configuration pattern, QED testers would have to either use *ad hoc* techniques of maintaining many different single instance configuration files for different operating contexts, such as different developers executing tests or running experiments on different hosts. CUTE, therefore, improves the adaptability, configurability, and scalability of T&E.

Pattern evaluation. In addition to the benefits of using the Variable Configuration pattern, the following is a list of benefits for using the Dynamic Variable Configuration pattern:

- It enables configurations to adapt to their operating environment. Developers do not have to have multiple configurations for each context, such as different hosts or users.
- It greatly increases the agility and configurability of a configuration for T&E.
- Developers can implement user-defined side-effects that understand the execution environment, such as auto-generating a UUID or binding to the appropriate network interface, and can be applied to the configuration. This helps increase the configuration flexibility for T&E.

Although the Dynamic Variable Configuration pattern has many benefits, it has several consequences. In addition to the consequences from the Variable Configuration pattern, the Dynamic Variable Configuration pattern has the following consequences:

- The configuration is *valid* if and only if C is valid and all side-effects in Δ execute successfully. This means developers are not able to learn about the validity of the configuration until it is evaluated in its target operating environment.

[2]The configuration in Listing VII.6 is invalid on standard Windows-based machines

- Even if each side-effect executes successfully, the result of each side-effect may produce an invalid value for its respective variable in the configuration. This can make the Dynamic Variable Configuration invalid.

VII.2.4 Batch Dynamic Variable Configuration Pattern

Problem statement. In Section VII.2.3 we discussed the Dynamic Variable Configuration pattern, which enables developers to determine portions of a configuation using context-based side-effects. Similar to the Variable Configuration pattern, the Dynamic Variable Configuration only produces a single instance configuration. When trying to *logically* group common configurations, it requires developers to rely on *ad hoc* techniques to realize the logical associations, such as grouping all configurations generated by each user based on tests that evaluate performance. Although this is acceptable in some cases, *e.g.*, testing the validity of a single instance configuration, it can negatively impact the scalability and maintainability of configurations for T&E.

Solution. The *Batch Dynamic Variable Configuration* pattern builds upon the Dynamic Variable Configuration pattern and enables large-scale component-based DRE system developers to logically group common configurations that are derived at the same time. Developers leverage the Batch Dynamic Variable Configuration pattern by defining logically related dictionaries, such as all the dictionaries used to generate configurations for evaluating security, in a single monolithic dictionary.

The Batch Dynamic Variable Configuration has the same formal definition of the Dynamic Variable Configuration (see Section VII.2.3) since it is using a single template. It, however, has a different evaluation function as illustrated by Equation VII.4:

$$DC'' = batcheval(C, D', \Delta) = \{\forall d \in D' : eval(eval(C, d), \Delta)\} \tag{VII.4}$$

where D' is the set of dictionaries such that if $d \in D'$, then d is an instance of D (see Section VII.2.1). DC'' is the set of configurations such that $C' = eval(C,d)$, $DC' = eval(C',\Delta)$, and $DC' \in DC''$.

Manifestation in CUTE and QED. We have realized the Batch Dynamic Variable Configuration pattern in CUTE and applied it to the QED project. QED testers leverage the Batch Dynamic Variable Configuration pattern using the following steps:

1. Define a template configuration file that contains variables and side-effects similar to the process in the Dynamic Variable Configuration pattern (see Section VII.2.3).

2. Define a set of dictionaries in a single file where each individual dictionary defines the key-value pair for each variable in the template configuration file.

Using the user-defined template file and batch configuration file, CUTE applies Equation VII.4 to produce a set of configuration files. QED testers then use the derived configuration files to evalate the system under development.

```
1  config (veryLowCPU.cdp) {
2    cpuTime=12.5
3    userName=‘/usr/bin/whoami‘
4  }
5
6  config (veryHighCPU.cdp) {
7    cpuTime=207.1
8    userName=‘/usr/bin/whoami‘
9  }
```

Listing VII.8: Batch dictionary for D&C that uses the Dynamic Variable Configuration pattern.

Listing VII.8 illustrates a set of dictionaries for the template in Listing VII.6. As highlighted in Listing VII.8, there are two different configurations named: `veryLowCPU.cdp` (line 1) and `veryHighCPU.cdp` (line 6). Likewise, Listing VII.9 shows the evaluation of the D&C excerpt for the batch configurations in Listing VII.8.

```
1  // veryLowCPU.cdp
```

```
...
  <configProperty>
    <name>cpuTime</name>
    <value>
      <type><kind>tk_long</kind></type>
      <value><long>12.5</long></value>
    </value>
  </configProperty>
  <configProperty>
    <name>testOwner</name>
    <value>
      <type><kind>tk_string</kind></type>
      <value><string>hillj</string></value>
    </value>
  </configProperty>
...

// veryHighCPU.cdp
...
  <configProperty>
    <name>cpuTime</name>
    <value>
      <type><kind>tk_long</kind></type>
      <value><long>207.1</long></value>
    </value>
  </configProperty>
  <configProperty>
    <name>testOwner</name>
    <value>
      <type><kind>tk_string</kind></type>
      <value><string>hillj</string></value>
    </value>
  </configProperty>
...
```

Listing VII.9: Evaluation of the Batch Variable Configuration pattern for the D&C.

Without the Batch Dynamic Variable Configuration pattern, QED testers would have a hard time logically maintaining and producing multiple configuration files used to evaluate system QoS. CUTE, therefore, helps increase the adaptability and configurability of T&E configurations.

Pattern evaluation. In addition to the benefits of the Dynamic Variable Configuration and Batch Variable Configuration, the following is a benefit for using the Batch Dynamic Variable Configuration pattern:

- Reduces the amount of single instance configurations needed to logically associate common configurations.

Although the Batch Dynamic Variable Configuration pattern has many benefits, it has several consequences. In addition to the consequences from the Dynamic Variable Configuration and Batch Variable Configuration Pattern, the Batch Dynamic Variable Configuration pattern has the following consequence:

- The batch configuration is *valid* if and only if all the individual configurations in the batch file are valid.

VII.3 Quantitative Analysis of Template Patterns

In this section, we quantify the that improvement testers gain for T&E from using each template pattern in the context of an example test scenario from the QED project.

VII.3.1 Revisiting the Multi-stage Workflow

In Chapter V Section V.3.1, we introduced the *multi-stage workflow* scenario as an example T&E scenario designed to test QED's ability to ensure application-level QoS properties, such as reliability and end-to-end response time, when handling applications with different priorities and privileges. To reiterate, the multi-stage workflow illustrated in Figure V.4 is composed of six different components. Each component in the multi-stage workflow contains behavior and workload, which can be configured by QED testers at D&C time using configuration files similar to the listings in Section VII.2. For example, developers can configure the periodicity of the `SensorClient` depending on what aspects of the QED they are testing (*e.g.*, using high periodicity to produce high utilizations) or alter the throughput ratio for an input event triggering an output event to produce different network workloads.

In addition, all experiments involving the multistage worklow application are conducted in ISISlab. Each development group on the QED project (*i.e.,* BBN Technologies, Boeing, and IMHC) uses the multistage workflow application to test their features in QED. This means T&E for the multistage workflow application must be able to handle a wide variety of configurations, operational scenarios, heterogeneity, such as different users executing the same test scenario under different ISISlab projects, or varying the CPU workload or event publish rate of each component to produce different effects on QED and the GIG middleware. Overall, the multistage workflow application provides QED testers with 11 points-of-variability for T&E.

If the QED testers relied on traditional techniques, such as handcrafted configuration files, then they would have a hard time managing T&E for the multistage workflow application. QED testers, therefore, leverage CUTE and its template patterns to increase the scalability and configurablity of the multistage workflow application. More importantly, they elect to use CUTE because CUTE will help them increase their scope of T&E and evaluate each point-of-variability. The remainder of this section quantifies the improvements that the template patterns manifested in CUTE provide QED testers.

VII.3.2 Quantitative Analysis of Results

To evaluate the improvement gained from CUTE, we calculate the number of single instance configuration files derived from each template pattern in Section VII.2, which is analogous to the number of *ad hoc* configuration files QED testers would have to manually create for T&E using traditional techniques. Although we discussed four template patterns in Section VII.2, the Batch Variable Configuration, Dynamic Variable Configuration, and

Batch Dynamic Variable Configuration pattern are defined in terms of the Variable Configuration pattern. The subtle difference is when testers realize the valid *range* of each variable, *i.e.*, the set of valid values for the variable.

We, therefore, can determine the number of single instance configuration files by analyzing the Variable Configuration pattern. Equation VII.5 highlights the equation for determining the number of single instance configuration files $|S|$—where S is the set of single instance configuration files—based on the points-of-variability in a single configuration (see Section VII.2.1).

$$|S| = \prod_{v \in V} |range(v)| \tag{VII.5}$$

As illustrated in Equation VII.5, $range(v)$ is the set of valid values for variable $v \in V$. The number of single instance configurations S that is realizable from a single template configuration, therefore, is determined by the product of the size in each variable's range, *i.e.*, $range(v)$. For example, using only a subset of the 11 points-of-variability in the multistage workflow application, if each component has a `cpuTime` integer variable (see Listing VII.1) that determines its CPU workload for the current test, and valid values for `cpuTime` are [10,100] msec, then QED testers can derive $(100-10) \times 6 = 540$ different single instance configurations from a single template configuration. Likewise, if there are 5 different QED testers executing the multistage workflow application in different ISISlab experiments and there is a dynamic variable named `userName=`/usr/bin/whoami`` to determine who is executing the test for accountability reasons, then the number of possible single instance configurations increases to $540 \times 5 = 2700$.

To QED testers, this is a great improvement over manually handcrafting 540 or 2700 different single instance configurations, especially when a large portion of the single instance configuration is constant (or static) between different configurations. Instead, QED testers focus on defining the different dictionaries used to instantiate the template configuration. This also helps concentrate QED testers efforts on locating points-of-variability in the T&E efforts, which can be used to conduct more controlled experiments against both QED and the GIG.

VII.4 Chapter Summary

Increasing T&E scope and scale can improve understanding and evaluation of large-scale component-based DRE system QoS concerns, such as performance, reliability, and security. This chapter presented four template patterns for improving T&E configurability and scalability. Each template pattern has been realized in the CUTE template engine. Testers use CUTE by defining configuration templates, and delay realization of single instance configuration until late in the system lifecycle, *e.g.*, when enough detail is known about the operating environment. Our analysis of CUTE on a representative large-scale component-based DRE project showed that it can significantly increase T&E scope without increasing the number of single instance configurations. Moreover, our quantitative analysis showed that CUTE can improve large-scale component-based DRE system QoS evaluation.

CHAPTER VIII

TECHNIQUE FOR PREDICTING END-TO-END RESPONSE TIME OF UNIQUE DEPLOYMENTS OF LOW UTILIZED SYSTEMS

Component-based technologies are raising the level of abstraction so software system developers of distributed real-time and embedded (DRE) systems can focus more on the application's "business-logic". Moreover, component-based technologies are separating many concerns, such as deployment (*i.e.*, placement of component on host) and configuration (*i.e.*, setting of component properties) [27], management of the application's lifecycle [29], and management of the execution environment's quality-of-service (QoS) policies [29, 110] of the application. The result of separating the concerns is the ability to fully address each concern independently of the application's "business-logic" (*i.e.*, its implementation).

Although component-based technologies are separating many concerns of DRE system development, realizing systemic (*system wide*) QoS properties, which is a key characteristic of DRE systems, is being pushed into the realized system's deployment and configuration (D&C) [27] solution space. For example, it is hard for component-based DRE system developers to fully understand systemic QoS properties, such as end-to-end response time of system execution paths until the system has been properly deployed and configured in its target environment. Only then would a component-based DRE system developer determine if different components designed to collaborate with each other perform better when located on different hosts compared to collocating them on the same host, which in turn might give rise to unforeseen software contentions, such as waiting for a thread to handle an event, or event/lock synchronization, or many known software design anti-patterns [112] in software performance engineering (SPE) [77].

Outside of brute force trial and error, conventional SPE techniques [60, 77, 129] can be used to evaluate different D&Cs. Although such techniques can assist in evaluating different D&Cs, existing techniques require manual construction of SPE models for each different deployment of the system. Unfortunately, this can be a daunting task as component-based DRE systems grow larger and more complex. Component-based DRE system developers, therefore, need improved techniques that reduce the complexity of evaluating systemic QoS properites of different D&Cs, such as end-to-end response time.

Solution approach: Use baseline profiles to evaluate deployments. Each individual component in a component-based DRE system has a baseline profile that captures its performance properties when deployed in isolation to other components. As components are collocated (*i.e.*, placed on the same host) with other components, their actual performance may deviate from its baseline performance profile due to unforeseen software contentions. Consequently, the component's deviation from its baseline performance not only affects its performance, but it also affects the performance of other components in the system, and systemic QoS properties, such as end-to-end response time.

This chapter describes our solution approach called the *Anti-deployment Solution sPace Analytical Model (Anti-SPAM)*, which predicts end-to-end response time of execution paths for different deployments of a component-based DRE system. Our approach uses baseline profiles for each component, which are obtained by instrumenting and profiling the components in a controlled environment. We then automatically construct an SPE analytical model for each different deployment of a component-based DRE system using a behavior graph of the system. Finally, we parameterize the workload of the auto-constructed SPE model and approximate the end-to-end response time of execution paths in the component-based DRE systems. This enables component-based DRE system developers to focus more on evaluating D&C's on their target architecture that are within acceptable error bounds, as opposed to evaluating each individual D&C.

Chapter organization. The remainder of this chapter is organized as follows: Section VIII.1 introduces a case study to highlight challenges of software contention; Section VIII.2 discusses our technique for evaluating unique deployments; Section VIII.3 validates our approach in the context of our case study; and Section VIII.4 summarizes the contributions of the chapter.

VIII.1 Case Study: The SLICE Shipboard Computing Scenario

In this section we use a representative case study from the shipboard computing environment domain to highlight how different D&Cs can introduce different software contention problems that impact systemic QoS properties.

VIII.1.1 Overview of the SLICE Scenario

The SLICE scenario is a representative example of a shipboard computing environment application. As illustrated in Figure VIII.1, the SLICE scenario consists of 7 different component instances, (*i.e.,* the rectangular objects): *SenMain*, *SenSec*, *PlanOne*, *PlanTwo*,

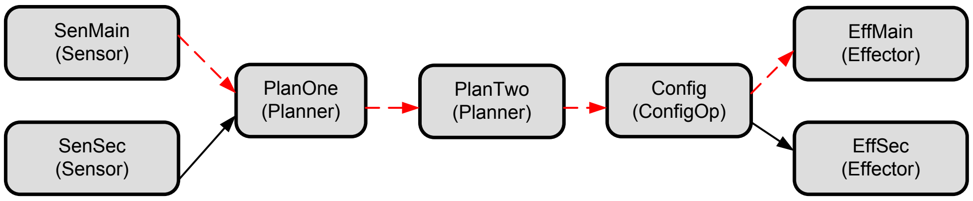


Figure VIII.1: High-level structural composition of the SLICE scenario

Config, *EffMain*, and *EffSec*. The directed lines between the components represent connections for inter-component communication, such as input/output events. The components in the SLICE scenario are deployed across 3 computing nodes and `SenMain` and `EffMain` are deployed on separate nodes to reflect the placement of physical equipment in the production shipboard environment. Finally, events that propagate along the path marked by the dashed lines in Figure VIII.1 represent an execution (or critical) path where meeting end-to-end response time within a specified exucution time is desirable.

VIII.1.2 Contention in the SLICE Scenario

We define software contentions as the deviation in performance as a result of the system's implementation—also characterized as performance anti-patterns [112]. In many cases, hardware contention does not have noticeable affects on resource utilization. For example, it is possible to have high end-to-end response time and low CPU utilization if different portions of the software components (*e.g.*, threads of execution) unnecessarily hold mutexes that prevent other portions of the software (or components) from executing. The (software) component holding the mutex may not be executing on the CPU, however, it can be involved in other activities, such as sending data across the network. Likewise, it is possible to have high response time if too many components are sending event requests to a single port of another component [96], which increases the queuing time for handling events.

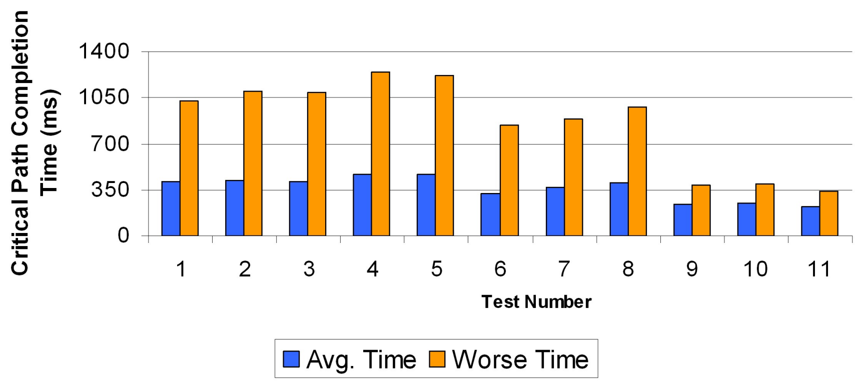


Figure VIII.2: Inital results of SLICE scenario to motivate software contention.

For example, Figure VIII.2 illustrates some initial results for ad hoc testing of the SLICE scenario in ISISLab to determine what unique deployments will have a end-to-end critical path response time of 350 msec. As shown in Figure VIII.2, each test respresents a different deployment and each deployment yields a different critical path end-to-end response time. For example, tests 10 and 11 have a critical path end-to-end response time difference of ~ 30 msec.

We believe this is due to unforeseen software contention resulting from collocating components that were designed and implemented in isolation to other components. More importantly, however, large-scale component-based DRE system developers would waste time and effort evaluating the performance of unique deployments that may not meet their performance needs, such as end-to-end response time. The remainder of this chapter, therefore, details how we use baseline profiles and SPE analytical models to understand the deployment solution space and evaluate end-to-end response time of component-based DRE systems.

VIII.2 Using Baseline Profiles to Evaluate Unique Deployments

This section discusses our technique for using baseline profiles to automatically construct SPE models to evaluate different deployments of component-based DRE systems.

VIII.2.1 Understanding the Unique Deployment Solution Space

After system composition, a component-based system becomes operational after its deployment and configuration in the target environment. There are, however, many ways to deploy a component-based systems, such as the SLICE scenario in Section VIII.1. For example, Table VIII.1 shows two unique deployments for the SLICE scenario where both differ by the placement of one component; the *SenMain* component on either Host1 or Host2.

When we consider all the unique deployments in the deployment solution space for a component-based systems, it is possible to represent such a space as a graph $G = (D, E)$ where:

- D is a set of vertices $d \in D$ in graph G that represents a unique deployment d in

Table VIII.1: Example of unique deployments in the SLICE scenario

	Deployment	
Host	A	B
1	SenMain, SenSec Config	SenSec, Config
2	EffSec, PlanOne	EffSec, PlanOne SenMain
3	PlanTwo, EffMain	PlanTwo, EffMain

the deployment solution space D. For example, deploying *SenMain* then *SenSec* on a single host H_i is the same as deploying *SenSec* then *SenMain* on the H_i. This is different from traditional bin packing, which is based on permutations [25].

- E is a set of edges $e \in E$ where $e_{i,j}$ is an edge between two unique deployments $i, j \in D$. Each edge e signifies moving a single component from one host to another host in the system.

Definition VIII.2.1. *If $C_{h,i}$ is the set of components deployed on host h in system H for deployment i, then a unique deployment is defined as:*

$$\exists h \in H : C_{h,i} \neq C_{h,j}; i, j \in D \wedge i \neq j \tag{VIII.1}$$

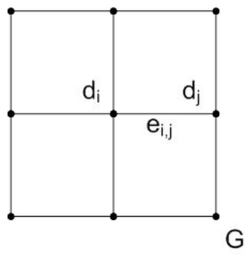


Figure VIII.3: Simple graph of component-based system deployment solution space

The resultant graph G, as shown in Figure VIII.3, allows us to create a visual representation of the deployment solution space. The visualization, however, has no concrete meaning from the QoS perspective because developers cannot understand how one deployment differs from another deployment except for the unique combination of components on each host. We address this problem by assigning a value to each edge e in graph G using Equation VIII.2,

$$val(e) = \delta \tag{VIII.2}$$

which gives rise to a directed graph G'. Figure VIII.4 illustrates G' where δ represents some difference of performance in a single QoS dimension, such as end-to-end response time or system throughput.

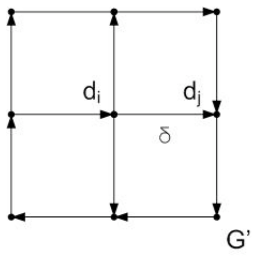


Figure VIII.4: Directed graph of component-based system deployment solution space

QoS, however, is known to comprise a N-dimensional space [100], and G' shown in Figure VIII.4 is the deployment solution space for a single dimension of QoS, *i.e.*, visualized on a single plane. When we consider N dimensions of QoS, we create an N-planar graph G''. Figure VIII.5 illustrates G'' where each plane represents a single QoS dimension. The value of each edge between unique deployments in different planes, *i.e.*, QoS dimensions, forms a N-tuple vector (δ_1, δ_2, ..., δ_n) where $\delta_i = val(e \in G'_i)$ and $i \in$ QoS dimension, $G'_i \in G''$.

When also considering the N-planer graph of the N QoS dimension solution space, an edge between the same deployment for each QoS dimension is governed by Property VIII.2.1. This results in a N-tuple vector equal to zero and is, therefore, ignored.

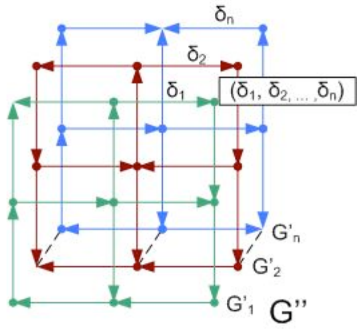


Figure VIII.5: N-planar directed graph of component-based system deployment solution space for N QoS dimensions

Property The value of an edge between the same unique deployment in different planes is zero.

Due to the complexity of the N-dimensional QoS solution space in relation to component-based system deployment, we focus on evaluating QoS in a single dimension in this chapter, *e.g.*, end-to-end response time. Moreover, even when evaluating QoS in a single dimension the solution space grows exponential as the number of hosts/components increase. The remainder of this section, therefore, discusses our technique for evaluating unique deployments using conventional SPE models with the goal of reducing the amount of testing on the target architecture.

VIII.2.2 Using Baseline Profiles to Analyze the Deployment Solution Space

To the best of our knowledge, this dissertation is the first research that investigates using SPE analyitcal techniques to construct analytical models that take both workload and deployment, or component placement, as input parameters to auto-construct SPE models. To reduce the complexity of the problem and gain better understanding of the problem, solution, and future research directions, we have limited the problem space to large-scale component-based systems to the category of M/M/1 models [77].

When we assume all communication is instantaneous, *i.e.*, neglect network workload, the end-to-end response of a large-scale component-based DRE that communicates asychronously using events can be predicted by Equation VIII.3

$$RT_P = \sum_i C_{i_{RT}} \tag{VIII.3}$$

where RT_P is the end-to-end response time of execution path P and $C_{i_{RT}}$ is the response time of a component, which is typically a port, in the execution path. In addition, since we are within the M/M/1 modeling class, we know the response time of a given component, or its individual port, is given by Equation VIII.4 [77]:

$$C_{i_{RT}} = \frac{1}{\mu - \lambda} \tag{VIII.4}$$

where $\mu = \frac{1}{S}$ is the service rate, given S is the service time, of the component's port and λ is the arrival rate of events into that particular component's port.

In Section VIII.1.2, we illustrated that different unique deployments have different end-to-end response times. Likewise, we also argued this is a result of contention experienced between collocated components. Agrawal [5], which was later reiterated in Menasce et al. [77], illustrate how you can elongate, or approximate, the service time in mixed models, such as open/closed models, that are independent of each other but compete for the same resources—similar to collocated components—based on the utilization of the other model using Equation VIII.5:

$$S' = \frac{S}{1 - U_{used}} \tag{VIII.5}$$

where S' is the elongated service time and U_{other} is the utilization of the other model. Using Equations VIII.3– VIII.5, Algorithm 2 lists our algorithm for automatically constructing a SPE analytical model using both component placement and workload as input parameters.

As illustrated in Algorithm 2, given the assembly of the system and its entry points,

Algorithm 2 General algorithm for auto-constructing an Anti-SPAM models

```
1:  procedure CONSTUCT($A, H, D, B, E$)
2:      $A$: set of components in the system and their interconnections
3:      $H$: set of hosts in the system
4:      $D$: mapping of components to hosts for A
5:      $B$: set of baseline service times for all components
6:      $E$: set of entry points into A
7:      A' = A
8:      for all do $e \in E$
9:          A' = propagate (A', $e_\lambda$, $A$)
10:     end for
11:     DM = $\emptyset$
12:     for all do $d \in D$
13:         DM = deploy (DM, d, A', H)
14:     end for
15:     AM = elongate (DM, B)
16:     return AM
17: end procedure
```

i.e., where events enter the system, we first propagate the arrival rate for each entry point through the assembly A, or system (line 9). After we propogate the arrival rates through the system, we construct a deployment model DM of the system (line 13) using the provided deployment information D for each component in the system. Given the deployment model of the system, we finally construct an analytical model AM by elongating the service times (line 15) using Equation VIII.5 where S is the baseline profile of component's port and U_{used} is the sum of the baseline utilization for the other collocated components, which is calculated using the propagated arrival rates and their corresponding baseline service times. Using the returned analytical model AM, which contains the predicted arrival rate λ and elongated service time S' for each component in the system, we then use Equation VIII.3 to predict the end-to-end response time for a given path, without having to manually construct an analytical model for each unique deployment of the system.

VIII.3 Evaluating the SLICE Scenario

In Section VIII.2, we presented a simple algorithm for predicting the end-to-end response time of unique deployments for component-based systems that communicate asynchronously using events. This section presents our results for validating our algorithm against the SLICE scenario to determine if we can predict the end-to-end response time within 10-15% error.

VIII.3.1 Experimental Setup

In Section VIII.1, we introduced the SLICE scenario. As explained in Section VIII.1, the SLICE scnerio is composed of 7 components that communicate via asynchrounous events. More importantly, the SLICE scenario contains a critical path and different unique deployments of the SLICE scenario yield different end-to-end response times for its critical path. We therefore want to predict the end-to-end response time of the critical path, which will help reduce the number of unique deployments that must be tested on the target architecture continuously throughout the software lifecycle, *i.e.*, test only the deployments that are within acceptable error bounds of their actual end-to-end response time.

To validate Anti-SPAM's end-to-end response time prediction capabilities, we used early integration testing techniques presented in Chapter III to model the behavior and workload of each component in the SLICE scenario. More specifically, we defined each component in the SLICE scenario such that it is bound to CPU workload only, since it is easy to emulate reliable CPU workload, and the CPU service time was randomly selected from the range of [10–90] msec. Likewise, we randomly selected the arrival rates of the events entering the system, *i.e.*, events that enter the system via `SenMain` and `SenSec`, from the range of [1–10] Hz. This enabled components in the SLICE scenario to obtain a baseline utilization in the range of [1–90]% when deployed in isolation.

After constructing the behavior and workload models for the SLICE scenario, we used the techniques presented in Chapter IV to generate emulation code for each component in the SLICE scenario that targeted the CIAO component-based middlware architecture. Likewise, we used the techniques in presented in Chapter VII to manage the complexity of generating many different deployments and configurations for testing.

Each deployment and configuration of the SLICE scnerio was executed in ISISlab (see Appendix A) for approximately 10 minutes. We chose to execute the system for 10 minutes because that was enough time for more than $\sim$ 600 events, which resulted in $\sim$ 1200 data points, to pass through the system and ensire the emulation of events arriving into the system reached the specified arrival rate for the given test. Finally, we used the techniques presented in Chapter V to analyzed data collected during each test run of the SLICE scenario.

VIII.3.2 Experimental Results

Initial Investigation. The main goal of Anti-SPAM is predict end-to-end response time of execution paths, such as the critical path in the SLICE scenario, within 10-15% error. Figure VIII.6 highlights our initial results for predicting the end-to-end response time of the critical path in the SLICE scenario. As illustrated in Figure VIII.6, the percent error between our predicted and measured results were not within our desired 10-15% error range.

Because the predicted end-to-end response time were not within our desired error range, we decided to investigate the problem in depth. First, we started with the percent error of the elongated service times since they are used to calculate the end-to-end response times. Table VIII.2 illustrates the analysis of a single unique deployment in the SLICE scenario. As highlighted in the Table VIII.2, the host(s) with the higher expected utilization, which is obtained by summing the baseline utilization of each component, contained components that had the higher percent errors in their elongated service times. To further understand the effects of expected host utilization on the elongated response times for each component,

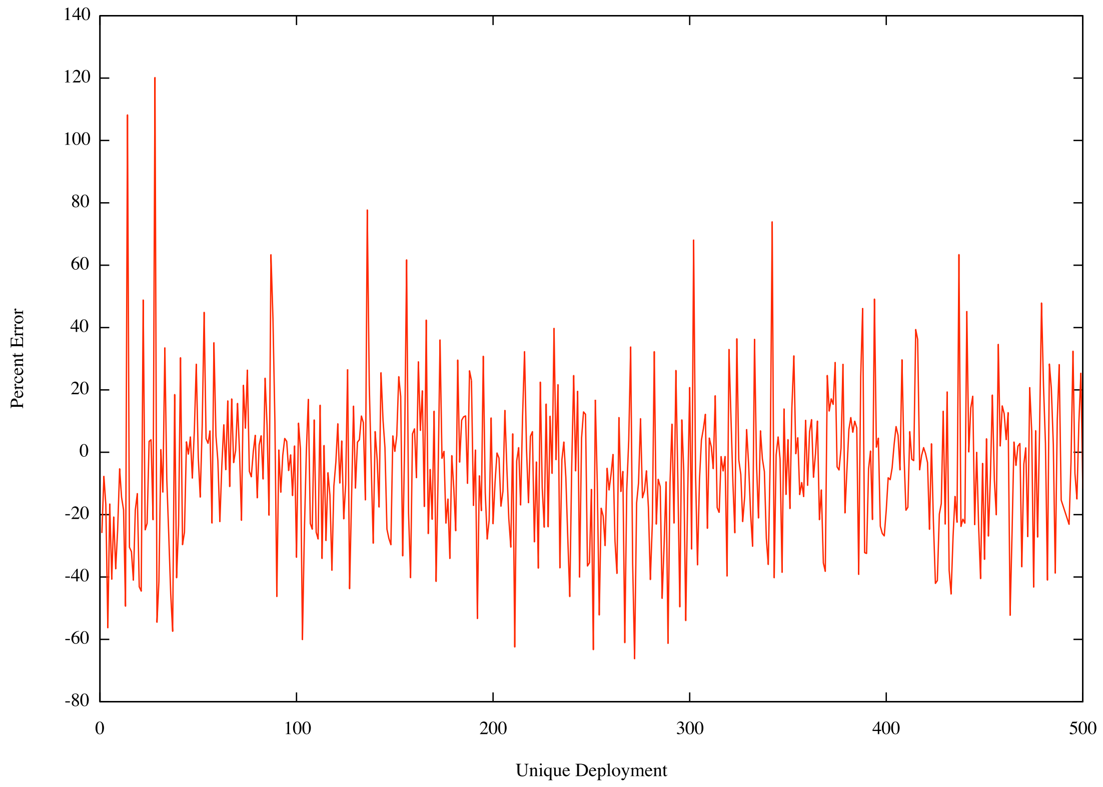


Figure VIII.6: Percentage error between initial measured and predicted end-to-end response time of SLICE critical path for unique deployments

Table VIII.2: Analysis of elongated service times for critical path components in a single deployment of the SLICE scenario

Instance	Measured (msec)	Predicted (msec)	Percent Error	Host Util.
SenMain	107.365	206.478	92.31	.753
PlanOne	19.829	12.58	-36.52	.126
PlanTwo	10.036	11.4416	14.01	.126
Config	149.27	56.68	-62.028	.753
EffMain	101.625	218.623	115.12	.753

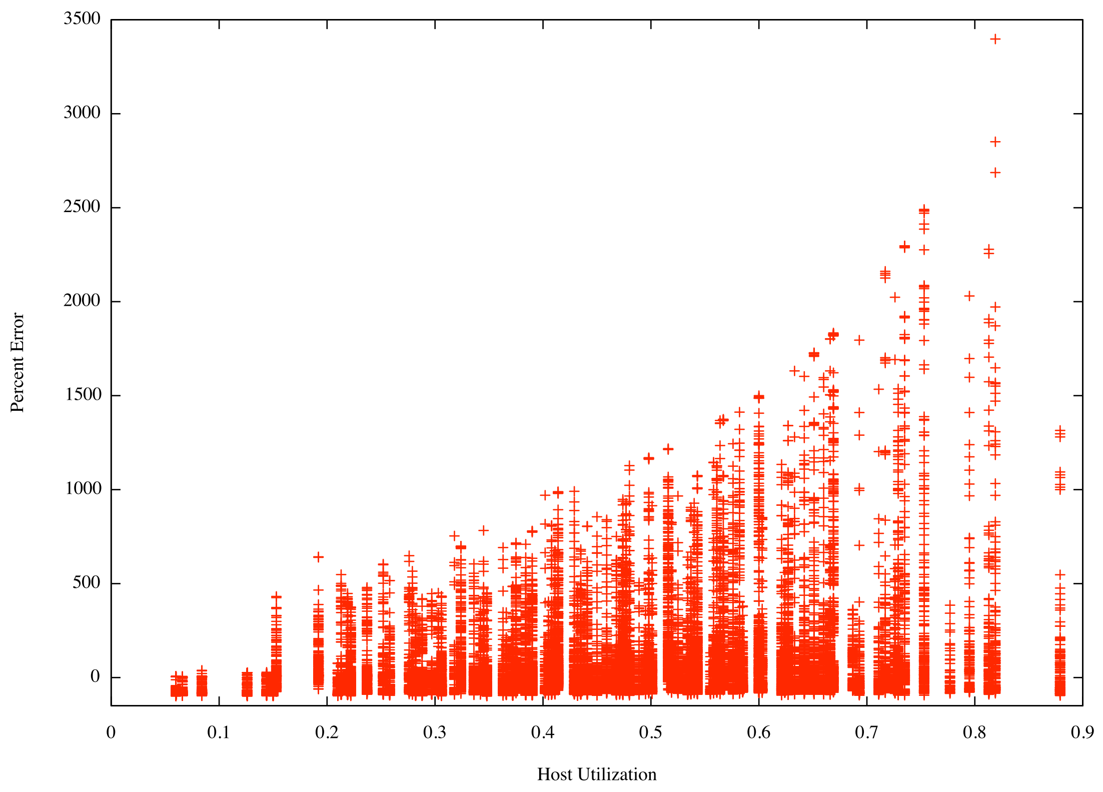


Figure VIII.7: Percent error in predicted vs. measured component response time error in relation to host utilization

Figure VIII.7 illustrates the percent error between the measure and predicted component response time in relation to the expected host utilization for 500 different unique deployments that were randomly selected. As illustrated in Figure VIII.7, as we add more components to a host, or increase its expected utilization, the more unstable and inconsistent the prediction becomes. Moreover, the more we increase the host's expected utilization, the greater the percent error.

Figure VIII.7 also provided some insights. As alluded to before, the more we increase the expected utilization, the greater the percent error is between expected versus meaured component response time. Figure VIII.7, therefore, helps us realize that if we bound the utilization of a host to low utilization, then the approximation equations in Section VIII.2 leveraged by Anti-SPAM should be within the 10-15% error bounds.

Retargeting Investigation. Using our newfound understanding of why the perdicted

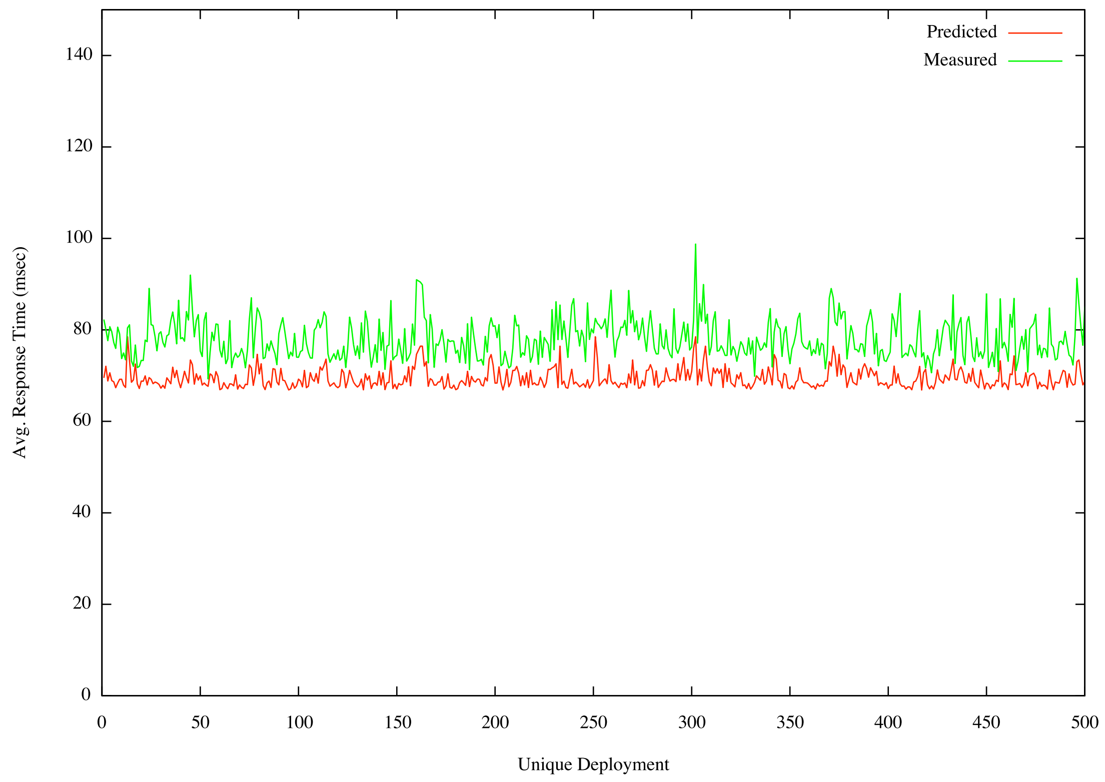


Figure VIII.8: Predicted vs. measured end-to-end response time of SLICE scenario under low utilization conditions

end-to-end response time may be greater than our desired 15% error, we retargeted our validation. This time we focused on low utilized system, such as systems currently experiencing off-peak hours of operation. We therefore redesigned our experiments such that the expected utilization of each host was no greater than 30% with collocated components, which is what would be considered low utilization [77].

Figure VIII.8 illustrates the measured versus predicted end-to-end response time for 500 randomly unique deployments of a single configuration under low host utilization. As illustrated in Figure VIII.8, the predictions for each unique deployment in the retargeted investigation was less than the measured end-to-end response time. Since our predictions were less than the measured end-to-end response time—and were more consistent—we used the average percent error to shift (or calibrate) the predicted end-to-end response time, which was approximately 6 msec, or 8%. Figure VIII.9 illustrates the calibrated predicted

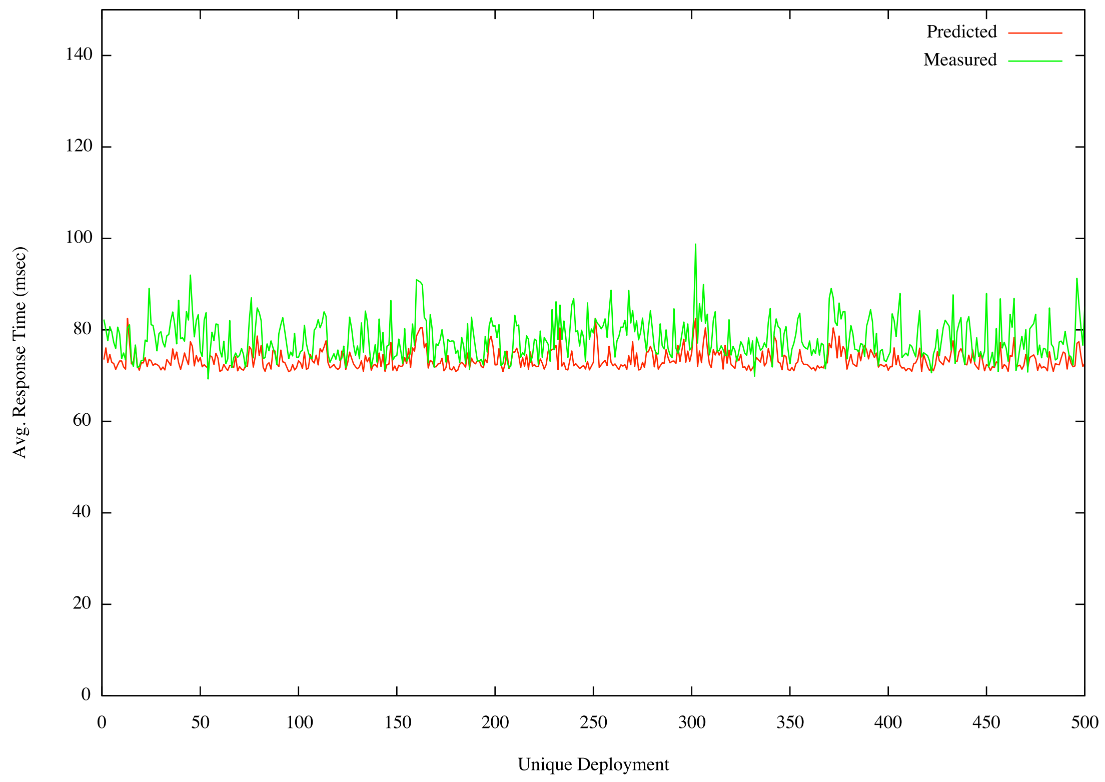


Figure VIII.9: Adjusted predicted vs. measured end-to-end response time of SLICE scenario under low utilizations

versus measured end-to-end response time for the configuration in Figure VIII.8 and two other configurations of the SLICE scenario. Likewise, Figure VIII.10 shows the percent error of the prediction for three different configurations of the SLICE scenario. As illustrated in Figure VIII.9, the predicted end-to-end response times are closer to the measured end-to-end response times. Figure VIII.10 also shows that our predictions (or approximations) are within a 10-15% error range. Likewise, the average error of in the approximation is $< 10\%$.

VIII.4 Chapter Summary

Poor deployment choices for a component-based system can have negative effects on systemic QoS properties, such as end-to-end response time. In this chapter, we presented a simple technique for predicting end-to-end response time of execution paths for different

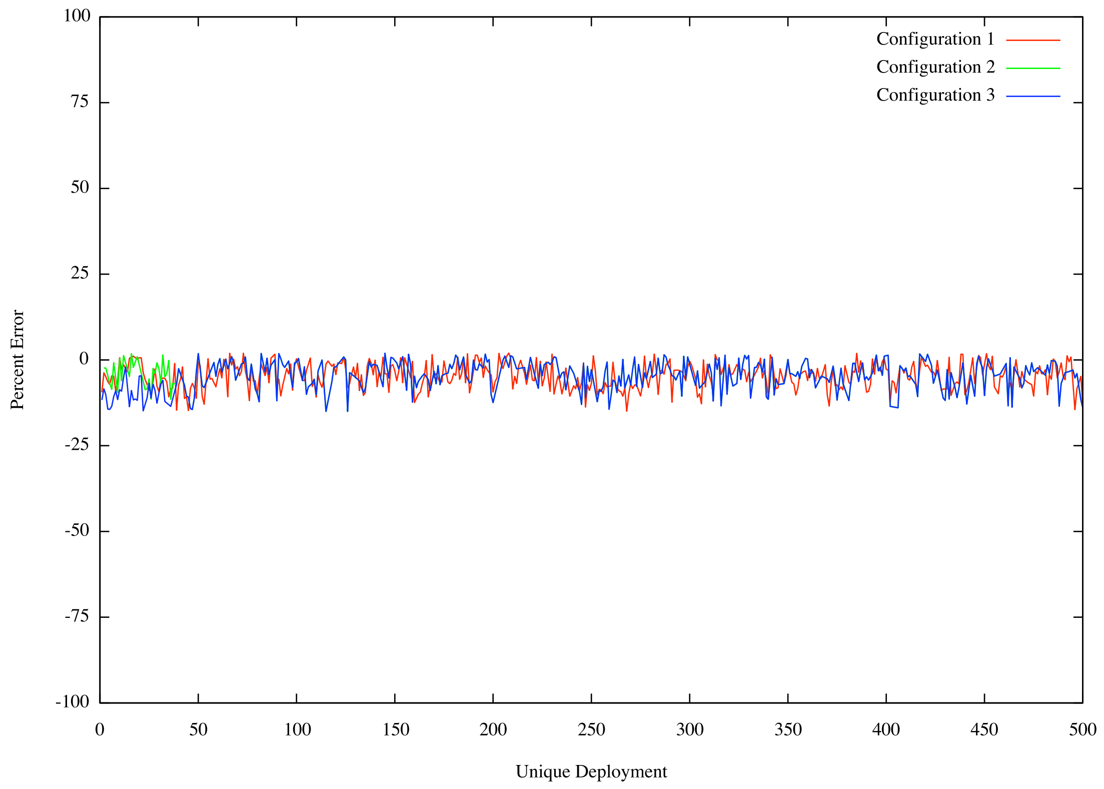


Figure VIII.10: Percent error of predicted end-to-end response time for multiple deployment and configurations

unique deployments. Our technique uses component location and workload, *i.e.*, baseline profiles of a component, as input parameters to automatically generate a SPE analytical model. We also applied our approach to predicting the end-to-end response time of a representative component-based DRE system that communicated using asynchrounous events assuming we classified the system as M/M/1. Moreover, we were able to predict the end-to-end response time of the representive component-based DRE systems critical path within 10-15% error when experiencing low utilizalation, such as off-peak hours of operation.

CHAPTER IX

CONCLUDING REMARKS

Component-based software engineering (CBSE) and agile development are addressing the functional concerns of development large-scale component-based distributed real-time and embedded (DRE) systems. Conventional agile development techniques, however, focus primarily on the functional concerns of such systems whereas evaluating quality-of-service (QoS) conerns is done during complete system integration time. Consequently, design flaws that negatively impact QoS are not discovered in a timely manner, and become costly to locate and rectify.

In this dissertation, we showed how DSML-based system execution modeling (SEM) tools can address the shortcomings of conventional agile development techniques, and bridge the gap between agile development and QoS evaluation of large-scale component-based DRE system. This is accomplished by (1) overcoming serailized-phasing development, (2) enabling emulation of the system under development on the target architecture for early QoS evaluation, and (3) doing so continuously throughout the software lifecycle. This dissertation also showed how DSML-based SEM tools can reduce the complexity of traditional software performance engineering (SPE) analytical models and evaluate end-to-end response time of a component-based DRE system where both deployment and workload are input parameters to the performance model. In the future, as large-scale component-based DRE systems become more complex and larger, such as ultra-large-scale systems [88] and systems-of-systems [58, 118], DSML-based SEM tools will provide an excellent software engineering approach for ensuring that both functional and QoS concerns are evaluated continuously throughout the software lifecycle, while reducing the cost of realizing such systems.

The following is a summary of lessons learned from the research work presented in this dissertation:

- Using a DSML based on a mathematical formalism to define behavior of components helps in specifying unambiguous behavior when generating code and configuration files for emulation and simulation.

- Separating the workload, behavior, and structural models allows each to evolve independently of each other. Moreover, it encourages the same behavior model to be supported in multiple structural models to increase portability, flexibility, and usability.

- Currently, we make the assumption that only a single event can be active on a behavior sequence. Multiple events can be active as long as each event represents a separate behavior sequence. Components, however, can have multiple events active in a behavior sequence depending on the number of threads active for an input event. We, therefore, need to extend semantic anchoring efforts to support the concept of multi-threaded input events.

- Using generative programming with templates that are parameterized by actions from behavior/workload models allows the DSML to easily adapt to different environments of execution.

- If changes to the DSML result in more points-of-generation, more points-of-visitation, or mutations to C++ artifacts (*e.g.,* adding/removing C++ classes that correspond to model elements), then each interpreter that reuses the same modified interpreter logic will be aware of these changes at compile-time and will need to be updated.

- If a DSML has only one use case, then single interpretation is the preferred implementation technique for its model interpreters. If multiple use cases have been identified, then the template metaprogramming technique is the preferred implementation technique for Visitor-based model interpreters.

- It is not hard to convert a single interpretation style interpreter to a template metaprogramming style interpreter because the main interpreter logic is already defined. It is "harder" to create a template metaprogramming interpreter from scratch because the different use cases for interpreters may not fully be understood *a priori*.

- Our template metaprogramming technique has greater benefit on model interpreters for complex DSMLs (*i.e.,* DSMLs with many elements and non-trivial parsing logic) than on model interpreters for simple DSMLs.

- C++ template metaprogramming incurs a steep learning curve for developers. When combined with a complex visitor hierarchy generated by modeling tools for a given modeling language, significant efforts must be expended to develop the interpreters. It is desirable for modeling frameworks to hide the complex visitor hierarchy but expose only the points-of-visitation and points-of-generation to the developers, which can enable rapid development of interpreters.

- Our experience applying UNITE to a representative component-based distributed system showed how it simplified identifying and extracting of metrics of interest for analyzing QoS concerns.

- UNITE help reduced the complexity of unit testing QoS concerns since it functioned independent of both data and system complexity.

- In many cases, only a subset of data extracted using UNITE needs to be processed, such as when looking for a particular value of interest. Future research therefore includes enabling support for parameterizing the causal relations to only extract data of interest.

- Before we had CiCUTS it was hard to produce and analyze large numbers of tests because developers and testers had to implement a custom testing framework to collect performance metrics and analyze the results manually. With CiCUTS, developers and testers could focus on resolving system performance issues instead of wrestling with low-level testing issues. Moreover, testing could occur at all hours of the day, especially during off-peak development hours (*e.g.*, from late at night to early morning) when the most testing resources were available.

- Prior to the creation of CiCUTS, we could not perform integration testing throughout the development phase. With CiCUTS, we could focus on improving the quality of RACE during its early stages of development instead of waiting until final integration time when the entire system (*i.e.*, infrastructure and application components) was complete.

- CiCUTS and UNITE use logging messages to collect performance metrics about the system being analyzed. Although this approach simplifies the collection process, it does not work well if the analyzed components (*e.g.*, third-party components available only in binary format) do not generate the necessary log messages. Moreover, it may be undesirable to augment source code with log messages because it may negatively impact system performance, especially in mission-critical DRE systems. In future work, we are integrating various interception techniques, such as dynamic instrumentation and analysis [17, 132], to capture metrics from such components transparently so they can be used within CiCUTS.

- Handcrafting template configurations for T&E can be labor intensive, especially if the template configurations are dense XML files. MDE techniques, such as domain-specific modeling languages, help alleviate the complexity of handcrafting such files via model interpreters that transform constructed models into concrete files. Our future work will integrate the template patterns in CUTE with MDE tools, such as GME, to improve the shortcomings of MDE tools and CUTE.

- Although CUTE can generate many different single instance configuration using a single template configuration, testers must manually run each configuration to evaluate large-scale component-based DRE system QoS. Continuous integration environments, such as CruiseControl (`cruisecontrol.sourceforge.net`) alleviate the complexity of manually executing tests via an autonomous build engine. Our future work will combine CUTE with continuous integration environments to improve the efficiency and effectiveness of running many tests continuously throughout the software lifecycle, especially when integrated with system execution modeling tools.

- Using deployment as an input parameter to the SPE analytical model allowed us to focus mainly on deployments that had execution paths that would meet desired end-to-end response time.

- Since Anti-SPAM is the first research approach to the best of our knowledge to use deployment as an input parameter into the model, we had to reduce the solution space to M/M/1 systems. Future work, therefore, includes relaxing this assumption so Anti-SPAM can evaluate the end-to-end response time of execution paths for many classes of systems.

- The approximiation technique in Anti-SPAM only works for systems experiencing low utilization. Future work, therefore, includes improving the approximation technique and heuristic so it can work for systems experiencing different levels of utilization.

- Anti-SPAM uses an approximation technique to evaluate the end-to-response time of an execution path. As we investigate the problem in more depth, we will begin to learn more about performance characteristics of such systems. Future work, therefore, includes not only understanding such performance characteristics of these systesm, but doing so from a theoretical standpoint, such as using fundanmental queueing theory principles.

The algorithms, analytics, patterns, and techniques described in this dissertation and realized in the CUTS SEM tool are available in open-source format at the following location: `http://www.dre.vanderbilt.edu/CUTS`.

APPENDIX A

ISISLAB

ISISlab (`www.isislab.vanderbilt.edu`) is an integration testbed at Vanderbilt University powered by Emulab software [101]. The Emulab software allows developers and testers to configure network topologies and operating systems on-the-fly to produce a realistic operating environment for distributed unit and integration testing. Figure A.1 shows a representative illustration of ISISlab at Vanderbilt University.

As illustrated in Figure A.1, each host in ISISlab is an IBM Blade Type L20, dual-CPU 2.8 GHz processor with 1 GB RAM. Each IBM Blade also contains four network interfaces. This allows complete separation of experiment-related network traffic from control-related network traffic, such as remotely connecting to a host via SSH and controlling the experiment. ISISlab is also configured with six Cisco 3750G-24TS network switches and one Cisco 3750G-48TS. This gives ISISlab a total of 192 Gigabit ports to use when configuring different network topologies for experimentation.

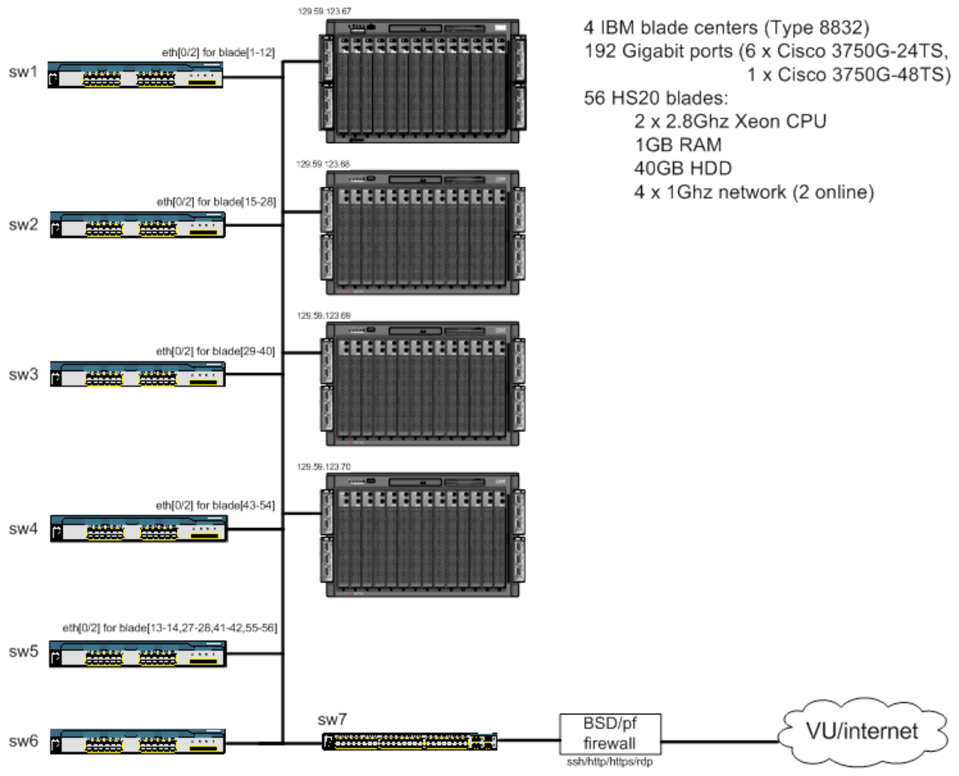

4 IBM blade centers (Type 8832)
192 Gigabit ports (6 x Cisco 3750G-24TS,
1 x Cisco 3750G-48TS)
56 HS20 blades:
2 x 2.8Ghz Xeon CPU
1GB RAM
40GB HDD
4 x 1Ghz network (2 online)
129.59.123.67
129.59.123.68
129.59.123.69
129.59.123.70
sw1
eth[0/2] for blade[1-12]
sw2
eth[0/2] for blade[15-28]
sw3
eth[0/2] for blade[29-40]
sw4
eth[0/2] for blade[43-54]
sw5
eth[0/2] for blade[13-14,27-28,41-42,55-56]
sw6
sw7
BSD/pf
firewall
ssh/http/https/rdp
VU/internet

Figure A.1: ISISlab at Vanderbilt University

REFERENCES

[1] BBN Technologies Awarded $2.8 Million in AFRL Funding to Develop System to Link Every Warfighter to Global Information Grid. BBN Technologies—Press Releases, www.bbn.com/news_and_events/press_releases/2008_press_releases/pr_21208_qed.

[2] Global Information Grid. The National Security Agency, www.nsa.gov/ia/industry/gig.cfm?MenuID=10.3.2.2.

[3] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[4] Aditya Agrawal, Tihamer Levendovszky, Jon Sprinkle, Feng Shi, and Gabor Karsai. Generative Programming via Graph Transformations in the Model-Driven Architecture. In *Workshop on Generative Techniques in the Context of Model Driven Architecture (OOPSLA 02)*, 2002.

[5] Subhash C. Agrawal. *Metamodeling: A Study of Approximations in Queueing Models*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[6] William A. Aiello, Yishay Mansour, S. Rajagopolan, and Adi Rosén. Competitive Queue Policies for Differentiated Services. *Journal of Algorithms*, 55(2):113–141, 2005.

[7] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994. Available from World Wide Web: `citeseer.ist.psu.edu/alur94theory.html`.

[8] Paolo Atzeni and Valeria De Antonellis. *Relational Database Theory*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

[9] Xiaoying Bai, Wei-Tek Tsai, Techeng Shen, Bing Li, and Ray Paul. Distributed End-to-End Testing Management. In *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, Seattle, WA, 2001.

[10] Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 190–199, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Don Batory. A Tutorial on Feature Oriented Programming and Product-lines. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 753–754. IEEE Computer Society, 2003.

[12] Jeffrey Bocarsly, Jonathan Harris, and Bill Hayduk. End-to-End Testing of IT Architecture and Applications. www.ibm.com/developerworks/rational/library/content/RationalEdge/jun02/EndtoEndTestingJun02.pdf, 2002.

[13] Egor Bondarev, Peter de With, and Michel Chaudron. Towards Predicting Real-Time Properties of a Component Assembly. In *Proceedings of the* 30th *EUROMICRO Conference*, pages 601–610, September 2004.

[14] Jon Bowyer and Janet Hughes. Assessing Undergraduate Experience of Continuous Integration and Test-driven Development. In *Proceeding of the 28th International Conference on Software Engineering (ICSE'06)*, pages 691–694, 2006.

[15] Don Box and Dharma Shukla. WinFX Workflow: Simplify Development with the Declarative Model of Windows Workflow Foundation. *MSDN Magazine*, 21:54–62, 2006.

[16] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, Reading, MA, 2003.

[17] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the General Track: 2004 USENIX Annual Technical Conference*, pages 15–28, June 2004.

[18] Kai Chen, Janos Sztipanovits, Sherif Abdelwahed, and Ethan K. Jackson. Semantic anchoring with model transformations. In *ECMDA-FA*, pages 115–129, 2005.

[19] Kai Chen, Janos Sztipanovits, and Sandeep Neema. Toward a Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 35–43, New York, NY, USA, 2005. ACM Press.

[20] Yih-Farn R. Chen, Emden R. Gansner, and Eleftherios Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Proceedings of the 6th European Conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 414–431, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[21] Roberta Coelho, Uirá Kulesza, Arndt von Staa, and Carlos Lucena. Unit Testing in Multi-agent Systems using Mock Agents and Aspects. In *International Workshop on Software Engineering for Large-scale Multi-agent Systems*, pages 83–90, 2006.

[22] Giulio Concas, Marco Di Francesco, Michele Marchesi, Roberta Quaresima, and Sandro Pinna. An Agile Development Process and Its Assessment Using Quantitative Object-Oriented Metrics. *Agile Processes in Software Engineering and Extreme Programming*, 9:83–93, 2008.

[23] Steve Cook, Gareth Jones, Stuart Kent, and Alan C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.

[24] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[25] Dionisio de Niz and Raj Rajkumar. Partitioning Bin-Packing Algorithms for Distributed Real-time Systems. *International Journal of Embedded Systems*, 2(3):196–208, 2006.

[26] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early Performance Testing of Distributed Software Applications. *ACM SIGSOFT Software Engineering Notes*, 29(1):94–103, January 2004.

[27] Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, and Aniruddha Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In *Proceedings of the 3rd Working Conference on Component Deployment (CD 2005)*, pages 67–82, Grenoble, France, November 2005.

[28] Gan Deng, Chris Gill, Douglas C. Schmidt, and Nanbor Wang. QoS-enabled Component Middleware for Distributed Real-Time and Embedded Systems. In I. Lee, J. Leung, and S. Son, editors, *Handbook of Real-Time and Embedded Systems*. CRC Press, 2007.

[29] Ada Diaconescu and John Murphy. Automating the Performance Management of Component-based Enterprise Systems Through the Use of Redundancy. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, pages 44–53, 2005.

[30] Bruce Powel Douglass. UML Statecharts. www-md.e-technik.uni-rostock.de/ma/gol/ilogix/umlsct.pdf.

[31] Marc Dutoo and Florian Lautenbacher. Java Workflow Tooling (JWT) Creation Review. www.eclipse.org/proposals/jwt/JWT%20Creation%20Review%2020070117.pdf, 2007.

[32] M.A. El-Gendy, A. Bose, and K.G. Shin. Evolution of the internet qos and support for soft real-time applications. *Proceedings of the IEEE*, 91(7):1086–1104, July 2003.

[33] Martin Fowler. Continuous Integration. www.martinfowler.com/articles/ continuousIntegration.html, May 2006.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[35] Aniruddha Gokhale, Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Arvind S. Krishna, George T. Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt. Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications. *The Journal of Science of Computer Programming: Special Issue on Foundations and Applications of Model Driven Architecture (MDA)*, 73(1):39–58, 2008.

[36] Swapna Gokhale, Aniruddha Gokhale, and Jeff Gray. A Model-Driven Performance Analysis Framework for Distributed, Performance-Sensitive Software Systems. In *Proceedings of the NSF NGS Workshop, International Conference on Parallel and Distributed Processing Symposium (IPDPS) 2005*, Denver, CO, April 2005.

[37] google-ctemplate. google-ctemplate. code.google.com/p/google-ctemplate, 2007.

[38] Jeff Gray, Juha-Pekka Tolvanen, Steven Kelly, Aniruddha Gokhale, Sandeep Neema, and Jonathan Sprinkle. Domain-Specific Modeling. In *CRC Handbook on Dynamic System Modeling, (Paul Fishwick, ed.)*, pages 7.1–7.20. CRC Press, May 2007.

[39] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, New York, 2004.

[40] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic Essence of AsmL. *Theoretical Computer Science*, 343(3):370–412, 2005.

[41] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. Available from World Wide Web: citeseer.ist.psu.edu/article/harel87statecharts.html.

[42] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *Software Engineering*, 16(4):403–414, 1990. Available from World Wide Web: citeseer.ist.psu.edu/harel90statemate.html.

[43] John Hatcliff, William Deng, Matthew Dwyer, Georg Jung, and Venkatesh Prasad. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of the 25th International Conference on Software Engineering*, pages 160–172, Portland, OR, May 2003.

[44] George T. Heineman and William T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[45] Scott Hissam, Gabriel Moreno, Judith Stafford, and Kurt Wallnau. Enabling Predictable Assembly. *Journal of Systems and Software*, 65(3):185–198, 2003.

[46] Jesper Holck and Niels Jorgenson. Continuous Integration and Quality Assurance: A Case Study of Two Open Source Projects. *Australasian Journal of Information Systems*, pages 40–53, 2003–2004.

[47] Chin-Yu Huang and Michael R. Lyu. Optimal Release Time for Software Systems Considering Cost, Testing-Effort, and Test Efficiency. *IEEE Transactions on Reliability*, 54(4):583–591, 2005.

[48] Dongping Huang and Hessam Sarjoughian. Software and Simulation Modeling for Real-Time Software-Intensive Systems. In *Proceedings of the Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'04)*, pages 196–203, Washington, DC, USA, 2004. IEEE Computer Society.

[49] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing in C# with NUnit*. The Pragmatic Programmers, Raleigh, NC, USA, 2004.

[50] Stephen D. Huston, James C. E. Johnson, and Umar Syyid. *The ACE Programmer's Guide*. Addison-Wesley, Boston, 2002.

[51] Software Engineering Institute. Ultra-Large-Scale Systems: Software Challenge of the Future. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, Jun 2006.

[52] Internet Engineering Task Force. Differentiated Services Working Group (diffserv) Charter. www.ietf.org/html.charters/diffserv-charter.html, 2000.

[53] David Janzen and Hossein Saiedian. Test-Driven Development: Concepts, Taxonomy, and Future Direction. *IEEE Computer*, 38(9):43–50, 2005.

[54] Nikolai Joukov, Timothy Wong, and Erez Zadok. Accurate and Efficient Replaying of File System Traces. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 25–25, 2005.

[55] Bjorn Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost*. Addison-Wesley Professional, September 2005.

[56] Gabor Karsai, Aditya Agrawal, Feng Shi, and Jonathan Sprinkle. On the Use of Graph Transformations in the Formal Specification of Computer-Based Systems. In *Proceedings of IEEE TC-ECBS and IFIP10.1 Joint Workshop on Formal Specifications of Computer-Based Systems*, pages 19–27, Huntsville, AL, USA, April 2003. IEEE.

[57] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata, Synthesis Lectures in Computer Science*. Morgan and

Claypool Publishers, San Rafael, CA, April 2006.

[58] Charles Keating, Ralph Rogers, Resit Unal, David Dryer, Andres Sousa-Poza, Robert Safford, William Peterson, and Ghaith Rabadi. System of Systems Engineering. *Engineering Management Journal*, page 36, September 2003.

[59] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[60] Samuel Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, 2006.

[61] Joel Kozikowski. A Bird's Eye View of AndroMDA. `galaxy.andromda.org/docs-3.1/contrib/birds-eye-view.html`.

[62] Ralf Lämmel, Eelco Visser, and Joost Visser. The Essence of Strategic Programming. 18 p.; Draft; Available at `http://www.cwi.nl/~ralf`, October15 2002.

[63] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[64] Kim Guldstrand Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In *FATES*, pages 79–94, 2004. Available from World Wide Web: `springerlink.metapress.com/openurl.asp?genre=article&amp;issn=0302-9743&amp;volume=3395&amp;spage=79`.

[65] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.

[66] Zhongjie Li, Wei Sun, Zhong Bo Jiang, and Xin Zhang. BPEL4WS Unit Testing: Framework and Implementation. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 103–110, Orlando, FL, 2005. IEEE Computer Society.

[67] Yan Liu, Alan Fekete, and Ian Gorton. Design-Level Performance Prediction of Component-Based Applications. *IEEE Transactions on Software Engineering*, 31(11):928–941, 2005.

[68] Yan Liu and Ian Gorton. Performance Prediction of J2EE Applications using Messaging Protocols. In *Proceedings of the International SIGSOFT Symposium Component-Based Software Engineering (CBSE)*, May 2005.

[69] Joseph Loyall, Marco Carvalho, Douglas Schmidt, Matthew Gillen, Andrew Martignoni III, Larry Bunch, James Edmondson, and David Corman. QoS Enabled Dissemination of Managed Information Objects in a Publish-Subscribe-Query Information Broker. In *Defense Transformation and Net-Centric Systems*, April 2009.

[70] Nancy Lynch and Mark Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[71] Qusay H. Mahmoud. *Middleware for Communications*. John Wiley and Sons, 2004.

[72] Daniela Mania, John Murphy, and Jennifer McManis. Developing Performance Models from Nonintrusive Monitoring Traces. *IT&T*, 2002. Available from World Wide Web: `citeseer.ist.psu.edu/541104.html`.

[73] Joan Mann. *The Role of Project Escalation in Explaining Runaway Information Systems Development Projects: A Field Study*. PhD thesis, Georgia State University, Atlanta, GA, 1996.

[74] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[75] Joe McKendrick. Ten companies where SOA made a difference in 2006. blogs.zdnet.com/service-oriented/?p=781, December 2006.

[76] Stephen J. Mellor, Marc J. Balcer, Stephen Mellor, and Marc Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley Professional, May 2002.

[77] Daniel A. Menasce, Lawrence W. Dowdy, and Virgilio A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[78] Microsoft Corporation. Microsoft .NET Development. `msdn.microsoft.com/net/`, 2002.

[79] Microsoft Corporation. Microsoft .NET Framework 3.0 Community. `www.netfx3.com`, 2007.

[80] Adrian Mos and John Murphy. Performance Monitoring of Java Component-Oriented Distributed Applications. In *IEEE 9th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 9–12, 2001.

[81] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-Driven Analysis and Synthesis of Concrete Syntax. In *International Conference on Modeling Driven Engineering and Languages Symposium (MoDELS 2006)*, pages 98–110, 2006.

[82] Anantha Narayanan and Gabor Karsai. Using Semantic Anchoring to Verify Behavior Preservation in Graph Transformations. *Electronic Communications of the EASST*, 4(2006), January 2006. Available from World Wide Web: chess.eecs.berkeley.edu/pubs/279.html.

[83] Margaret Naughton, James McGrath, and Donal Heffernan. Real-time Software Modelling using Statecharts and Timed Automata Approaches. In *Proceedings of the IEE Irish Signals and Systems Conference*, Dublin, Ireland, June 2006.

[84] Norman Neff. Attribute Based Compiler Implemented Using Visitor Pattern. In *Proceedings of the 35th Technical Symposium on Computer Science Education (SIGCSE 04)*, pages 130–134, New York, NY, USA, 2004. ACM Press.

[85] Dung "Zung" Nguyen, Mathias Ricken, and Stephen Wong. Design Patterns for Parsing. In *Proceedings of the 36th Technical Symposium on Computer Science Education (SIGCSE 05)*, pages 477–481, New York, NY, USA, 2005. ACM Press.

[86] Iftikhar Azim Niaz. Code Generation From Uml Statecharts. Available from World Wide Web: citeseer.ist.psu.edu/635920.html.

[87] Steven Nordstrom, Shweta Shetty, Di Yao, Shikha Ahuja, Sandeep Neema, and Ted Bapty. The Action Language: Refining a Behavioral Modeling Language. In *Proceedings of the 12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005)*, Piscataway, NJ, USA, 2005. IEEE.

[88] L. Northrop, P. Feiler, R. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. *Ultra-Large-Scale Systems: Software Challenge of the Future*. Software Engineering Institute, Pittsburgh PA, 2006.

[89] Object Management Group. *Lightweight CCM RFP*, realtime/02-11-27 edition, November 2002.

[90] Object Management Group. *Deployment and Configuration Adopted Submission*, OMG Document mars/03-05-08 edition, July 2003.

[91] Object Management Group. *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 edition, May 2003.

[92] Object Management Group. BPMN Information Home. www.bpmn.org, 2005.

[93] Object Management Group. *CORBA Components v4.0*, OMG Document formal/2006-04-01 edition, April 2006.

[94] openArchitectureWare. openArchitectureWare. www.openarchitectureware.org, 2007.

[95] Trevor Parsons, Adrian, and John Murphy. Non-Intrusive End-to-End Runtime Path Tracing for J2EE Systems. *IEEE Proceedings Software*, 153:149–161, August 2006.

[96] Trevor Parsons and John Murphy. *Detecting Performance Antipatterns in Component Based Enterprise Systems*. PhD thesis, University College Dublin, Belfield, Dublin 4, Ireland, 2007.

[97] 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.

[98] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[99] Runtao Qu, Satoshi Hirano, Takeshi Ohkawa, Takaya Kubota, and Radu Nicolescu. Distributed Unit Testing. Technical Report CITR-TR-191, University of Auckland, 2006.

[100] Raghunathan Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. A Resource Allocation Model for QoS Management. In *In Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

[101] Robert Ricci, Chris Alfred, and Jay Lepreau. A Solver for the Network Testbed Mapping Problem. *SIGCOMM Computer Communications Review*, 33(2):30–44, April 2003.

[102] Linda Rising and Norman S. Janoff. The Scrum Software Development Process for Small Teams. *IEEE Software*, 17(4), 2000.

[103] Robby, Matthew Dwyer, and John Hatcliff. Bogor: An Extensible and Highly-Modular Model Checking Framework. In *Proceedings of the* 4^{th} *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, Helsinki, Finland, September 2003. ACM.

[104] Nilabja Roy, Akshay Dabholkar, Nathan Hamm, Larry Dowdy, and Douglas Schmidt. Modeling Software Contention using Colored Petri Nets. In *16th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Baltimore, MD, September 2008.

[105] David Saff and Michael D. Ernst. Reducing Wasted Development Time via Continuous Testing. In *Proceedings of* $14^t h$ *International Symposium on Software Reliability Engineering*, pages 281–292, November 2003.

[106] David Saff and Michael 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.

[107] Douglas C. Schmidt. Adaptive and Reflective Middleware for Distributed Real-time and Embedded Systems. In *EMSOFT 2001: First Workshop on Embedded Software.*, Lake Tahoe, CA, October 2001.

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

[109] Markus Schordan. The Language of the Visitor Design Pattern. *Journal of Universal Computer Science*, 12(7):849–867, 2006.

[110] Nishanth Shankaran, Xenofon Koutsoukos, Douglas C. Schmidt, and Aniruddha Gokhale. Evaluating Adaptive Resource Management for Distributed Real-time Embedded Systems. In *Proceedings of the 4th Workshop on Adaptive and Reflective Middleware*, Grenoble, France, November 2005.

[111] Nishanth Shankaran, Douglas C. Schmidt, Yingming Chen, Xenofon Koutsoukous, and Chenyang Lu. The Design and Performance of Configurable Component Middleware for End-to-End Adaptation of Distributed Real-time Embedded Systems. In *Proc. of the 10th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2007)*, Santorini Island, Greece, May 2007.

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

[113] Connie U. Smith and Lloyd G. Williams. Performance Engineering Evaluation of Object-Oriented Systems with SPE*ED. In *Proceedings of the 9th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, pages 135–154, London, UK, 1997.

[114] Connie U. Smith and Lloyd G. Williams. Software Performance Antipatterns. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 127–136, 2000.

[115] Suzanne Smith and Sara Stoecklin. What We Can Learn from Extreme Programming. *Journal of Computing Sciences in Colleges*, 17(2):144–151, 2001.

[116] A. Snow and M. Keil. The Challenges of Accurate Project Status Reporting. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, 2001.

[117] Software Composition and Modeling (Softcom) Laboratory. Constraint-Specification Aspect Weaver (C-SAW). www.cis.uab.edu/ gray/Research/C-SAW, University of Alabama, Birmingham, AL.

[118] Andrés Sousa-Poza, Samuel Kovacic, and Charles Keating. System of Systems Engineering: an Emerging Multidiscipline. *International Journal of System of Systems Engineering (IJSSE)*, 1:1–17, 2008.

[119] SUN. Java Messaging Service Specification. `java.sun.com/products/jms/`, 2002.

[120] Sun Microsystems. Enterprise JavaBeans Specification. java.sun.com/products/ejb/docs.html, August 2001.

[121] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, April 1997.

[122] Toufik Taibi, Lim Boon Ping, Ng Sheau Wen, Lai Kiat Sing, and Chew Keow Lim. Developing a Distributed Stock Exchange Application using CORBA. In *Proceeding of the Student Conference on Research and Development (SCOReD)*, Putraiaya, Malaysia, 2003.

[123] The Mathworks Inc. Simulink/Stateflow. `www.mathworks.com/products/simulink`.

[124] Juha-Pekka Tolvanen. MetaEdit+: Domain-specific Modeling for Full Code Generation. In *OOPSLA/GPCE '04: Companion to the 19th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 39–40, New York, NY, 2004. ACM.

[125] CodeSmith Tools. CodeSmith Tools. `www.codesmithtools.com`, 2009.

[126] Mauro Tortonesi, Cesare Stefanelli, Niranjan Suri, Marco Arguedas, and Maggie Breedy. Mockets: A Novel Message-Oriented Communications Middleware for the Wireless Internet. In *International Conference on Wireless Information Networks and Systems (WINSYS 2006)*, August 2006.

[127] Bruce Trask and Angel Roman. Model Driven Engineering Basics using Eclipse. In *Proceeding of ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Genova, Italy, October 2006.

[128] Wei-Tek Tsai, Xiaoying Bai, Ray J. Paul, Weiguang Shao, and Vishal Agarwal. End-To-End Integration Testing Design. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, pages 166–171, Chicago, IL, October 2001.

[129] Alexander Ufimtsev and Liam Murphy. Performance Modeling of a JavaEE Component Application using Layered Queuing Networks: Revised Approach and a Case Study. In *Proceedings of the Conference on Specification and Verification of Component-based Systems (SAVCBS '06)*, pages 11–18, 2006.

[130] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[131] Todd L. Veldhuizen. Five Compilation Models for C++ Templates. In *First Workshop on C++ Template Programming*, Erfurt, Germany, October 2000.

[132] Daniel G. Waddington, Nilabja Roy, and Douglas C. Schmidt. Dynamic Analysis and Profiling of Multi-threaded Systems. In Pierre F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*. Idea Group, 2007.

[133] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz, and Christopher D. Gill. QoS-enabled Middleware. In Qusay Mahmoud, editor, *Middleware for Communications*, pages 131–162. Wiley and Sons, New York, 2004.

[134] Elaine J. Weyuker. Testing Component-based Software: A Cautionary Tale. *Software, IEEE*, 15(5):54–59, Sep/Oct 1998.

[135] Jules White, Andrey Nechypurenko, Egon Wuchner, and Douglas C. Schmidt. Reducing the Complexity of Optimizing Large-scale Systems by Integrating Constraint Solvers with Graphical Modeling Tools. In Pierre F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*. Idea Group, 2007.

[136] Jules White and Douglas C. Schmidt. Simplifying the Development of Product-line Customization Tools via Model Driven Development. In *Proceedings of the MODELS 2005 workshop on MDD for Software Product-lines*, Half Moon Bay, Jamaica, October 2005.

[137] Jules White, Douglas C. Schmidt, and Aniruddha Gokhale. Simplifying autonomic enterprise java bean applications via model-driven engineering and simulation. *Journal of Software and System Modeling*, 7(1):3–23, 2008.

[138] Ye Wu, Mei-Hwa Chen, and Jeff Offutt. UML-Based Integration Testing for Component-Based Software. In *Proceedings of the Second International Conference on COTS-Based Software Systems*, pages 251–260. Springer-Verlag, 2003.

[139] Hany F. EL Yamany, Miriam A. M. Capretz, and Luiz F. Capretz. A Multi-Agent Framework for Testing Distributed Systems. In *30th Annual International Computer Software and Applications Conference*, pages 151–156, 2006.