CiCUTS: Combining System Execution Modeling Tools with Continuous Integration Environments

James H. Hill, Douglas C. Schmidt
Vanderbilt University
Nashville, TN, USA
{j.hill, d.schmidt}@vanderbilt.edu

Adam A. Porter
University of Maryland
College Park, MD, USA
aporter@cs.udm.edu

John M. Slaby
Raytheon Company
Portsmouth, RI, USA
john_m_slaby@raytheon.com

Abstract

System execution modeling (SEM) tools provide an effective means to evaluate the quality of service (QoS) of enterprise distributed real-time and embedded (DRE) systems. SEM tools facilitate testing and resolving performance issues throughout the entire development life-cycle, rather than waiting until final system integration. SEM tools have not historically focused on effective testing. New techniques are therefore needed to help bridge the gap between the early integration capabilities of SEM tools and testing so developers can focus on resolving strategic integration and performance issues, as opposed to wrestling with tedious and error-prone low-level testing concerns.

This paper provides two contributions to research on using SEM tools to address enterprise DRE system integration challenges. First, we evaluate several approaches for combining continuous integration environments with SEM tools and describe CiCUTS, which combines the CUTS SEM tool with the CruiseControl.NET continuous integration environment. Second, we present a case study that shows how CiCUTS helps reduce the time and effort required to manage and execute integration tests that evaluate QoS metrics for a representative DRE system from the domain of shipboard computing. The results of our case study show that CiCUTS helps developers and testers ensure the performance of an example enterprise DRE system is within its QoS specifications throughout development, instead of waiting until system integration time to evaluate QoS.

1 Introduction

Challenges of developing enterprise distributed real-time and embedded (DRE) systems. Enterprise DRE systems, such as supervisory control and data acquisition (SCADA) systems, air traffic control systems, and shipboard computing environments, are increasingly being developed using service-oriented architectures (SOAs), such as the Lightweight CORBA Component Model (CCM), Microsoft .NET, and J2EE. SOAs address certain software development challenges, such as reusing core application logic or improving application scalability and reliability [19]. They still, however, often incur unresolved problems, such as "serialized phasing" [11], where application level components are not tested until long after infrastructure level components.

System execution modeling (SEM) tools [5, 8, 11] are a promising technology for addressing serialized phasing problems of SOA-based enterprise DRE systems. SEM tools help to (1) capture computational workloads, resource utilizations and requirements, and communication of application and infrastructure-level components using domain-specific modeling languages [15], (2) emulate the captured system specifications on actual infrastructure components and hardware in production runtime environments to determine their impact on QoS properties, such as measuring the performance impact of dynamically managing shipboard computing environment resources, and (3) process QoS metrics and provide informative analysis to enhance the system's architecture and components to improve QoS.

Although SEM tools assist with developing enterprise DRE systems, they historically focused on performance analysis [22] rather than efficient testing. Moreover, due to the inherit complexities of serialized phasing (*e.g.*, portions of the system being developed in phases defer QoS testing until integration testing) it is hard to use SEM tools continuously to test for QoS throughout the entire development lifecycle. New techniques are therefore needed to enhance the capabilities of existing SEM tools to improve their testing capabilities throughout the entire development lifecycle.

Solution Approach → Integrate SEM tools with continuous integration environments. Continuous integration [9] environments, such as CruiseControl (`cruisecontrol.sourceforge.net`), Build Forge (`www.buildforge.com`), and DART (`public.kitware.com/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.

This paper describes the design of *CiCUTS*, which combines the CruiseControl.NET continuous integration environment the Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS) [11], which is a SEM tool that enables developers to evaluate system QoS in the target domain before system integration time. CiCUTS tests are deployed into the target domain to enable system QoS testing. These tests are managed by CruiseControl.NET and dictate the behavior of CUTS, as shown in Figure 1. This process is

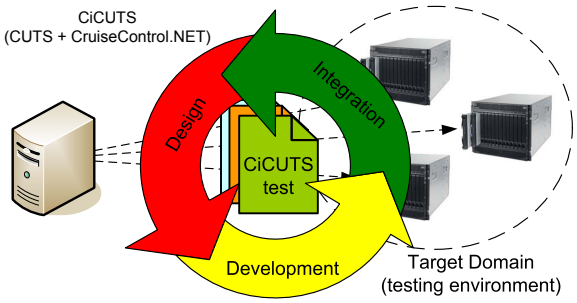

Figure 1. CiCUTS: Combining SEM Tools with Continuous Integration Environments

repeated continuously to help developers and testers ensure system QoS meets—or is close to meeting—its specification throughout the development lifecycle.

This paper also describes how we applied CiCUTS to a representative DRE system case study from the domain of shipboard computing. The results from our case study show how combining continuous integration environments with SEM tools enabled CiCUTS 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.

Paper organization. The remainder of this paper is organized as follows: Section 2 introduces our enterprise DRE system case study from the domain of shipboard computing; Section 3 describes how CiCUTS combines SEM tools with a continuous integration environment; Section 4 shows how we used CiCUTS to evaluate the QoS of our case study system; Section 5 compares our work on CiCUTS with related work; and Section 6 presents concluding remarks and lessons learned.

2 The RACE DRE System Case Study

This section describes a SOA-based enterprise DRE system case study from the domain of shipboard computing to motivate the need for, operation of, and benefits provided by CiCUTS. This case study is based on the *Resource Allocation and Control Engine* (RACE) [21], which is an open-source distributed resource manager system we developed using the CIAO [7] implementation of the Lightweight CORBA Component Model (CCM) [18] 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 [21].

2.1 Overview of RACE

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

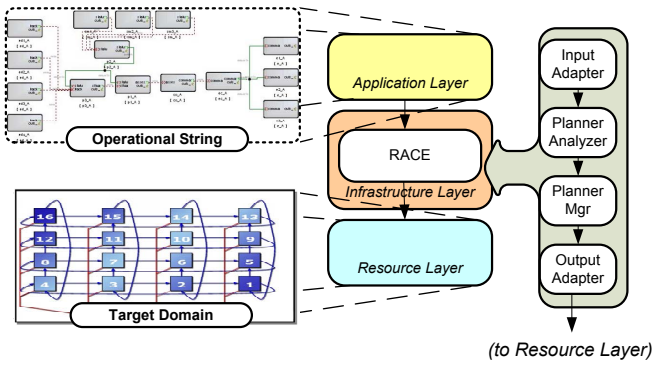

Figure 2. Architecture of RACE

strategies—*static* and *dynamic*—for DRE systems. 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.

2.2 RACE's Baseline Scenario

The case study in this paper 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 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 problem outlined in Section 1. 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. Our prior experience [11] with distributed resource management services indicated that deferring QoS testing until final system integration requires much more effort to rectify the inevitable performance problems uncovered during integration testing.

To avoid the problems outlined above, we used CUTS [11] 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 paper describes CiCUTS and the results of our experiments that apply it to evaluate RACE's baseline scenario throughout its development lifecycle.

3 Combining System Execution Modeling Tools with Continuous Integration Environments

This section discusses the design of CiCUTS and explains how it enables continuous system QoS testing from design-time to system integration time.

3.1 Overview of CiCUTS

CiCUTS is a combination of the CruiseControl.NET continuous integration environment 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 [17] and non-intrusive [16, 20] 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 1), which are NAnt subtasks that drive CUTS and the testing environment to create realistic scenarios (Requirement 2 in Section 1). CruiseControl.NET then continuously manages and executes CiCUTS tests throughout the development lifecycle (Requirement 1 in Section 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 of Section 1).

3.2 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.

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 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.

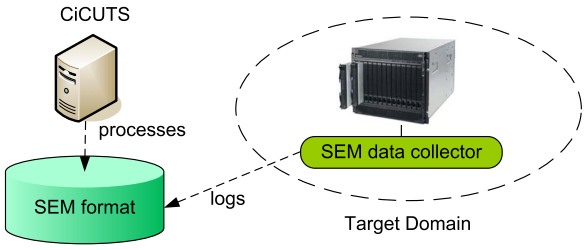

Figure 3. Conceptual model of design alternative 1

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.

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.

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 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.

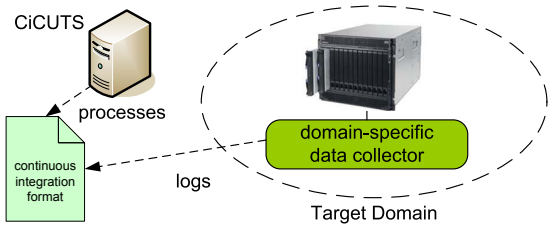

Figure 4. Conceptual model of design alternative 2

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

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 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 environments format

Evaluation. This approach applies the Bridge pattern [10] to capture metrics in an intermediate format, such as XML or a BLOB in a centralized database, that is neither

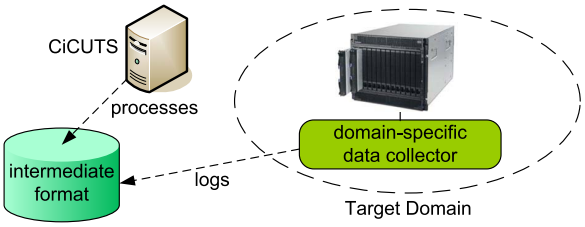

Figure 5. Conceptual model of design alternative 3

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 tester 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 [13], 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 [4,12] and support extension for domain-specific needs, such as evaluating the success of results generated by domain-specific extensions.

3.3 The Structure and Functionality of CiCUTS

After evaluating the pros and cons of the various approaches described above, we chose 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 were redeveloped using a different SOA technology (*e.g.*, Microsoft.NET or J2EE), CruiseControl.NET could still be applied since it operates on the intermediate format, not the SEM tools's format. Moreover, testers and developers can use any data collection technique specific to the target SOA technology, such as the Java Messaging Service [24] for J2EE applications, as long as 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.

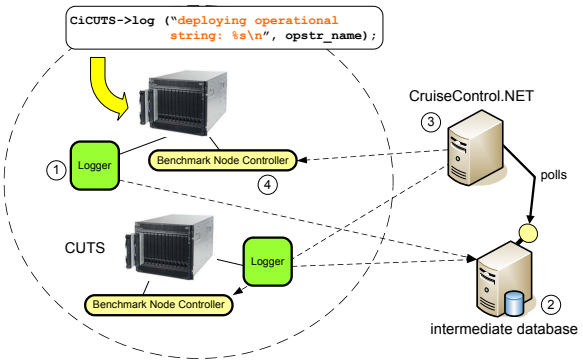

Figure 6. Structure of CiCUTS

Figure 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 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 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 in Section 1). Finally, testers instruct CruiseControl.NET to manage and execute the CiCUTS tests (*i.e*, Requirement 1 in Section 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 in Section 1).

4 Experiment Results Using CiCUTS to Evaluate System QoS

This section shows the design and results of experiments that applied CiCUTS to evaluate the QoS of RACE's *baseline scenario* described in Section 2.2. 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.

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 7 shows a PICML[1] model for one of the baseline scenario's operational strings—which was replicated 10 times to create the 10 operational strings in the baseline scenario—consisting of 15 interconnected components represented by the rounded boxes.

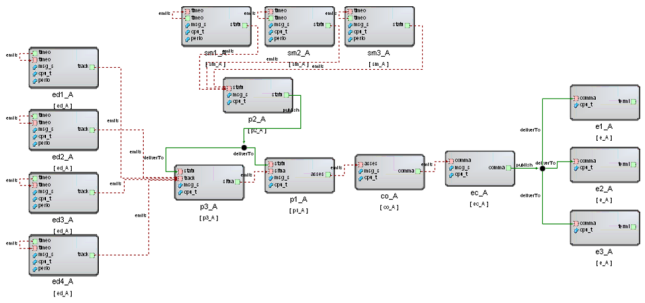

Figure 7. Graphical Model of the Replicated Operational String for the Baseline Scenario

The four components on the left side of the operational string in Figure 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 7 are system observation components that monitor the state of the system. The four linear components in the bottom-center of Figure 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 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 3.3), we used PICML to construct the 10 operational strings described above. We then used the CUTS SEM tool portion of CiCUTS to generate Lightweight CCM compliant emulation code that represented each component in the operational string managed by RACE (see Figure 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 specified in Table 1. The importance values[2] 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 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 3.1 to collect information, such as time of operational string deployment/teardown or time of node failure recognition.

Table 1. Importance Values of the Baseline Scenario Operational Strings

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

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 un-

[1] The Platform Independent Component Modeling Language (PICML) [2] is a domain-specific modeling language for modeling compositions, deployments, and configurations of Lightweight CCM applications.

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

der 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 8) to show whether the lifetime of dynamic deployments exceed the lifetime of static deployments based on resource availability with respect to environmental changes.

4.2 Experiment Results

This section presents the results of experiments that validate H1 and H2 about CiCUTS (see Section 4) when evaluating the QoS of the RACE baseline scenario.

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

H1 conjectured that CiCUTS will assist in understanding the behavior and performance of infrastructure-level applications, such as RACE, well before system integration. Figure 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 Figure 8—which are specific to

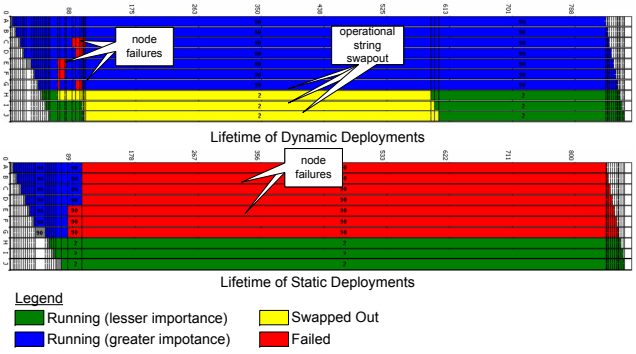

Figure 8. Graphical Analysis of Static Deployments (bottom) vs. Dynamic Deployments (top) using RACE

RACE—were generated from the log messages stored in the database via the CUTS loggers described in Section 3.2. 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 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 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 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 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 2). If the failed nodes were recovered, however, RACE would attempt to redeploy the lower importance operational strings. Figure 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.

4.2.2 Using CiCUTS to Ensure Performance is Within QoS Specifications

H2 conjectured 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 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 9 highlights the maximum number of tests we captured from the baseline scenario presented in Figure 8 after it was executed approximately 427 times over a 2 week period. The

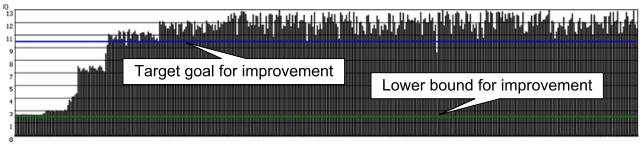

Figure 9. Overview Analysis of Continuously Executing the RACE Baseline Scenario

number of executions corresponds to the number of times a modification (such as a bug fix or an added feature to RACE) was detected in the source code repository at 30 minute intervals.

The vertical bars in Figure 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 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 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 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 9, each set of modifications within a predefined time period (*e.g.*, every 30 minutes) corresponded to a single tests 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.

5 Related Work

This section compares our work on CiCUTS with other related works on continuous integration environments and SEM tools.

5.1 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 [4] et al discuss their experience using continuous integration environments to assess undergraduates experience using test-driven development (TDD) [14]. CiCUTS extends their effort by evaluating different techniques for integrating continuous integra-

tion environments with external processes/tools. Moreover, CiCUTS 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 [4], only releasing modules after they pass all automated test cases [23], and reducing integration risk by finding errors earlier in the lifecycle [12]. Our work on CiCUTS also shows the benefits of using continuous integration environments to automate key aspects of the testing process. CiCUTS, 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.

5.2 End-to-end Performance Testing

The design and application of an end-to-end integration test suite for J2EE is discussed in [1, 25]. Although CiCUTS also focuses on end-to-end integration testing, it combines continuous integration environments and SEM tools instead of implementing a custom environment for end-to-end system integration. CiCUTS also extends the work in [1, 25] by focusing on automating the execution of a large number of tests to increase the fidelity of the QoS results, whereas [1, 25] focus on validating functional correctness.

Real-time Technology Solutions (RTTS) [3] discusses how to achieve end-to-end testing of applications by testing at the component-level and system level throughout development. CiCUTS is similar to RTTS in that it validates if an analyzed system satisfies its functional and performance requirements. Likewise, both RTTS and CiCUTS achieve end-to-end testing by combining preexisting testing tools, such as SEM tools and continuous integration environments. CiCUTS, however, extends the RTTS work by focusing on an environment for systematically executing a large numbers of tests to validate system QoS.

6 Concluding Remarks

This paper described the design and application of CiCUTS, which combines the CUTS system execution modeling (SEM) tools with the CruiseControl.NET continuous integration environment. We evaluated the design alternatives we considered when integrating CUTS with CruiseControl.NET and explained the structure and functionality of the approach we selected. We also presented a case study that applied CiCUTS to a representative enterprise 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 CruiseControl.NET's continuous integration capabilities to enhance its testing capabilities. Moreover, instead of spending time implementing a custom testing framework, we focused on developing test scenarios that systematically exercised various static and dynamic deployment capabilities of RACE. To improve our quantitative analysis of RACE we used CiCUTS to generate and analyze a large number of test results during a short period of time as RACE evolved. We were therefore able to ensure RACE was within its QoS specifications throughout its development lifecycle, rather than waiting until final system integration when it would be harder to resolve problems related to RACE meeting its QoS specifications.

Based on our experience devising and running the RACE experiments using CiCUTS, we learned the following lessons:

- **CiCUTS improved quantitative testing of performance requirements.** 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.
- **CiCUTS improved qualitative testing of performance requirements.** 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 lacks support for third-party component development.** CiCUTS uses logging messages to collect performance metrics about systems it is analyzing. 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 [6, 26], to capture metrics from

such components transparently so they can be used within CiCUTS.

CiCUTS and RACE are available in open-source format and can be downloaded from `www.dre.vanderbilt.edu/CUTS` and `www.dre.vanderbilt.edu/CIAO`, respectively.

References

[1] X. Bai, W.-T. Tsai, T. Shen, B. Li, and R. Paul. Distributed End-to-End Testing Management. In *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, Seattle, WA, 2001.

[2] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. 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, Los Alamitos, CA, USA, 2005.

[3] J. Bocarsly, J. Harris, and B. Hayduk. End-to-End Testing of IT Architecture and Applications. `www.ibm.com/developerworks/rational/library/content/RationalEdge/jun02/EndtoEndTestingJun02.pdf`, 2002.

[4] J. Bowyer and J. 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.

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

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

[7] G. Deng, C. Gill, D. C. Schmidt, and N. 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.

[8] M. Dutoo and F. Lautenbacher. Java Workflow Tooling (JWT) Creation Review. www.eclipse.org/proposals/jwt/JWT2007.

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

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[11] J. H. Hill, J. Slaby, S. Baker, and D. C. Schmidt. Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.

[12] J. Holck and N. Jorgenson. Continuous Integration and Quality Assurance: A Case Study of Two Open Source Projects. *Australasian Journal of Information Systems*, pages 40–53, 2003–2004.

[13] S. D. Huston, J. C. E. Johnson, and U. Syyid. *The ACE Programmer's Guide*. Addison-Wesley, Boston, 2002.

[14] D. Janzen and H. Saiedian. Test-Driven Development: Concepts, Taxonomy, and Future Direction. *Computer*, 38(9):43–50, 2005.

[15] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[16] D. Mania, J. Murphy, and J. McManis. Developing Performance Models from Nonintrusive Monitoring Traces. *IT&T*, 2002.

[17] D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[18] Object Management Group. *Lightweight CCM RFP*, realtime/02-11-27 edition, Nov. 2002.

[19] C. O'Hanlon. A conversation with Werner Vogels. *Queue*, 4(4):14–22, 2006.

[20] T. Parsons, Adrian, and J. Murphy. Non-Intrusive End-to-End Runtime Path Tracing for J2EE Systems. *IEEE Proceedings Software*, 153:149–161, August 2006.

[21] N. Shankaran, D. C. Schmidt, Y. Chen, X. Koutsoukous, and C. 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.

[22] C. U. Smith and L. 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.

[23] S. Smith and S. Stoecklin. What We Can Learn from Extreme Programming. *Journal of Computing Sciences in Colleges*, 17(2):144–151, 2001.

[24] SUN. Java Messaging Service Specification. `java.sun.com/products/jms/`, 2002.

[25] W.-T. Tsai, X. Bai, R. J. Paul, W. Shao, and V. 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.

[26] D. G. Waddington, N. Roy, and D. C. Schmidt. Dynamic Analysis and Profiling of Multi-threaded Systems. In P. F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*. Idea Group, 2007.