Preprint for *1999 International Symposium on Software Reliability Engineering, Boca Raton, FL.*

Robustness Testing of A Distributed Simulation Backplane

Kimberly Fernsler
RS6000 Graphics Development
IBM Corporation
Austin, TX 78758
kimfern@us.ibm.com

Philip Koopman
Institute for Complex Engineered Systems &
Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
koopman@cmu.edu

Abstract

Creating robust software requires not only careful specification and implementation, but also quantitative measurement. This paper describes Ballista exception handling testing of the High Level Architecture Run-Time Infrastructure (HLA RTI). The RTI is a standard distributed simulation system intended to provide completely robust exception handling, yet implementations have normalized robustness failure rates as high as 10%. Non-robust testing responses include exception handler crashes, segmentation violations, "unknown" exceptions, and task hangs. Other issues include different robustness failure modes across ports to two operating systems, and mandatory client machine rebooting after a particular RTI failure. Testing the RTI led to scalable extensions of the Ballista architecture for handling exception-based error reporting models, testing object-oriented software structures (including callbacks, pass by reference, and constructors), and operating in a state-rich, distributed system environment. These results demonstrate that robustness testing can provide useful feedback to high-quality software development processes, and can be applied to domains well beyond the previous work on testing operating systems.

1. Introduction

Robustness in software is becoming essential as critical computer systems increasingly pervade our daily lives. It is not uncommon (and although annoying, usually not catastrophic) for desktop computing applications to crash on occasion. However, as more and more software applications become essential to the everyday functioning of our society, we are entering an era in which software crashes are becoming unacceptable in an increasing number of application areas.

Careful specification and implementation are required to create robust software (*i.e.,* software that responds gracefully to overload and exception conditions [11]). In particular, it is thought by some that exception handling is a significant source of operational software failures [3]. However, cost, time, and staffing constraints often limit software testing to the important area of functional correctness, while leaving few resources to determine a software system's robustness in the face of exceptional conditions.

The Ballista software robustness testing service provides a way for software modules to be automatically tested and characterized for robustness failures caused by exception handling failures. This service provides a direct, repeatable, quantitative method to evaluate a software system's robustness. Ballista works by performing tests on the software based on traditional "black box" testing techniques (*i.e.,* behavioral rather than structural testing) to measure a system's responses when encountering exceptional parameter values (overload/stress testing is planned as future work). Previously the focus of Ballista was on testing the robustness of several implementations of the POSIX [12] operating system C language Application Programming Interface (API), and found a variety of robustness failures that included repeatable, complete system crashes that could be caused by a single line of source code [15].

This paper explores whether the Ballista testing approach works on an application area that is significantly different than an operating system API, testing the hypothesis that Ballista is a general-purpose testing approach that is scalable across multiple domains. The new application area selected for testing is the Department of Defense's High Level Architecture Run-Time Infrastructure (HLA RTI). The RTI is a general-purpose simulation backplane system used for distributed military simulations, and is specifically designed for robust exception handling. The RTI was chosen as the next step in the development of Ballista because it not only has a significantly different implementation style than a C-language operating system API, but also because it has been intentionally designed to be very robust, and should ideally have no robustness failures. HLA has been designed to be part of a DoD-wide effort to establish a common technical framework to facilitate the interoperability and reuse of all types of models and simulations, and represents the highest priority effort within the DoD modeling and simulation community [4]. Because RTI applications are envisioned to consist of large numbers of software modules integrated from many different vendors, robust operation is a key concern.

Testing the RTI involved stretching the Ballista architecture to address exception-based error reporting models, test object-oriented software structures (including callbacks), incorporate necessary state-setting "scaffolding" code, and operate in a state-rich distributed system environment. Yet, this expansion of capabilities was accomplished with minimal changes to the base Ballista architecture.

Beyond demonstrating that the Ballista approach applies to more than one domain, the results of RTI testing themselves yield insights into the types of problems that can occur even with an application designed to be highly robust. Testing the RTI revealed a significant number of unhandled exception conditions, unintended exceptions, and processes that can be made to "hang" in the RTI. Additionally, problems were revealed in providing equivalent exception handling support when the RTI was ported to multiple platforms, potentially undermining attempts to design robust, portable application programs.

In the remainder of this paper, Section 2 discusses how Ballista works and what extensions were required to address the needs of RTI testing. Section 3 presents the experimental methodology. Section 4 presents the testing results, and Section 5 provides conclusions and a discussion of future work.

2. Extending Ballista for RTI testing

Ballista testing works by bombarding a software module with combinations of exceptional and acceptable input values. The reaction of the system is measured for either catastrophic failure (generally involving a machine reboot), a task "hang" (detected by a timer), or a task "abort" (detected by observing that a process terminates abnormally). The current implementation of Ballista draws upon lists of heuristic test values for each data type in a function call parameter list, and executes combinations of these values for testing purposes. In each test case, the function call under test is called a single time to determine whether it is robust when called with a particular set of parameter values.

2.1. Prior work

The Ballista testing framework is based on several generations of previous work in both the software testing and fault-tolerance communities. The Crashme program and the University of Wisconsin Fuzz project are both prior examples of automated robustness testing. Crashme works by writing random data values to memory and then attempts to execute them as code by spawning a large number of concurrent processes [2]. The Fuzz project injects random noise (or "fuzz") into specific elements of an OS interface [16]. Both methods find robustness problems in operating systems, although they are not specifically designed for a high degree of repeatability, and Crashme in particular is not generally applicable for testing software other than operating systems.

Approaches to robustness testing in the fault tolerance community are usually based on fault injection techniques, and include Fiat, FTAPE, and Ferrari. The Fiat system modifies the binary image of a process in memory [1]. Ferrari, on the other hand, uses software traps to simulate specific hardware level faults, such as errors in data or address lines [13]. FTAPE uses hardware-dependent device drivers to inject faults into a system running with a random load generator [21]. These approaches have produced useful results, but were not intended to provide a scalable, portable quantification of robustness for software modules.

There are several commercial tools such as Purify [20], Insure++ [19], and BoundsChecker [18] that test for robustness problems by instrumenting software and monitoring execution. They work by detecting exceptions that arise during development testing of the software. However, they are not able to find robustness failures in situations that are not tested. Additionally, they require access to source code, which is not necessarily available. In contrast, Ballista testing works by sending selected exceptional and acceptable input combinations directly into already-compiled software modules at the module testing level. Thus, Ballista is different from (and potentially complementary to) instrumentation-based robustness improvement tools.

The Ballista approach has been used to compare the robustness of off-the-shelf operating system robustness by automatically testing each of 15 different implementations of the POSIX[12] operating system application programming interface (API). The results demonstrated that there is significant opportunity for increasing robustness within current operating systems [22]. Questions left unanswered from the operating system studies were whether other APIs might be better suited to robust implementations, and whether the Ballista approach would work well in other application domains. This paper describes progress in answering those questions.

2.2. General Ballista robustness testing

Ballista actively seeks out robustness failures by generating combinational tests of valid and invalid parameter values for system calls and functions. Rather than base testing on the behavioral specification of the function, Ballista instead uses only data type information to generate test cases. Because in many APIs there are fewer distinct data types than functions, this approach tends to scale test development costs sub-linearly with the number of functions to be tested. Additionally, an inheritance approach permits reusing test cases from one application area to another.

As an example of Ballista operation, Figure 1 shows the actual test values used to test the RTI function `rtiAmb.subscribeObjectClassAtributes`, which takes parameters specifying an `RTI::ObjectClassHandle` (which is type-defined to be an `RTI::ULong`), an `RTI::AttributeHandleSet`, and an `RTI::Boolean`. The fact that this particular RTI function takes three parameters of three different data types leads Ballista to draw test values from three separate test objects, each established for one of the three data types. For complete testing, all combinations of test values are used, in this case yielding 25 ULongs * 14 AttributeHandleSets * 12 Booleans = 4200 tests for this function (statistical sampling of combinations can be used for very large test sets, and has been found to be reasonably accurate in finding failure rates compared to exhaustive testing).

A robustness failure is defined within the context of Ballista to be a test case which, when executed, produces a non-robust reaction such as a “hang”, a core dump, or generation of an illegal/undefined exception within the software being tested. In general, this corresponds to an implicit functional specification for all software modules being tested of “doesn’t crash the computer, doesn’t hang, and doesn’t abnormally terminate.” This very general functional specification is the key to minimizing the need for per-function test development effort, because all functions are considered to have a single identical functional specification -- the actual computation performed by any particular function is ignored for robustness testing purposes.

2.3. Enhancements for RTI testing

The previously tested POSIX operating systems represent only a small fraction of the types and variations of COTS software that could potentially benefit from robustness testing. So, a big question in the past has been whether a testing methodology initially developed using an example application of operating system testing would actually work in a different domain. Testing the RTI with Ballista did in fact require extensions to incorporate exception-based error reporting models,

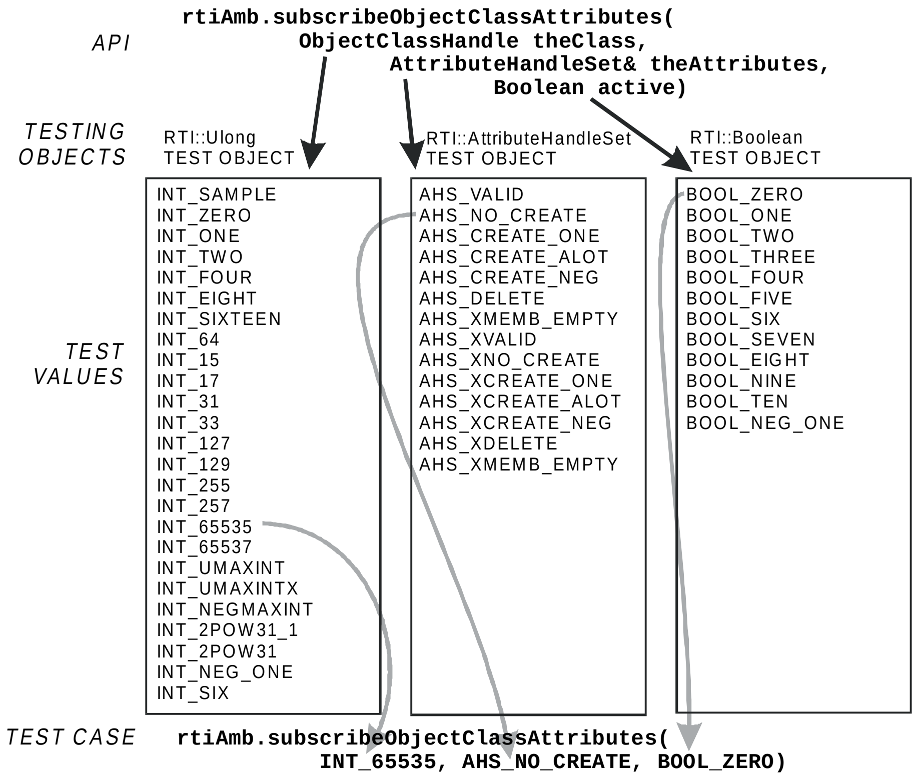

Figure 1. A Ballista example RTI test case for a function that allows the federate to subscribe to a set of object attributes.

testing object-oriented software structures (including callbacks, passing objects by reference, class data types, private class member data, and constructors), addition of test scaffolding code, and operating with a state-rich, distributed software framework.

In previous Ballista testing, any call which resulted in a signal being thrown was considered a robustness failure by the test harness. However, the RTI throws an RTI-defined exception rather than using the POSIX strategy of return codes. This was readily accommodated by including user-defined exception handling code in the general Ballista testing harness. Thus, any user-defined exception was considered a "pass" except for the "unknown" RTI exception, which indicated an internal RTI exception handling software defect.

Because the RTI is a distributed system, a certain amount of setup code must be executed to set the distributed system state before a test can be executed. While in the operating system testing all such "scaffolding" was incorporated into constructors and destructors for each test value instance (such as creating a file for a read or write operation), in the RTI there were some function-specific operations required to create reasonable test starting points. While at first it seemed that distinct scaffolding would be required to test each and every RTI function, it turned out that we were able to group the RTI functions into 10 equivalence classes, with each class able to share the same test scaffolding code. This was incorporated into Ballista by inserting optional user-configurable setup and shutdown code to be applied before and after each test case, enabling clean set up and shutdown of the RTI environment for each specific test performed. Thus, while scaffolding code was required for testing the RTI, the amount of code and development effort was relatively small.

The RTI specification requires that some RTI function calls be able to support a defined callback function. In a typical RTI simulation execution, there are many other simulation processes which need to communicate and share data with each other. Testing the RTI showed that the Ballista framework is flexible enough to support the RTI callbacks with essentially no changes.

In addition, the RTI contains object oriented features such as passing by reference, user-defined class data types, constructors, and private class member data. In general these were handled in relatively simple ways requiring little or no change to the test harness. Perhaps the most difficult situation that arose was how to test a function that took a pass-by-reference parameter of a class rather than an actual object. This problem was resolved by creating a pointer to the class as the data type, and modifying Ballista slightly to accommodate the syntax for pointers in its test definition language. In general, all the problems that had previously seemed to be large obstacles by the development team and external reviewers alike compared to testing operating systems were resolved with similarly trivial adjustments to the testing system.

3. Experimental methodology

The current Ballista implementation uses a portable testing client that is downloaded and run on a developer's machine along with the module under test. The client connects to the Ballista testing server at Carnegie Mellon that directs the client's testing of the module under test. This service allows software modules to be automatically and rapidly tested and characterized for robustness failures, and was particularly useful for testing RTI robustness on multiple platforms. Testing the RTI on Digital Unix and SunOS only required recompiling the small Ballista client on each target machine, avoiding the need to port the server-side software.

3.1. Interfacing to the RTI for testing

The HLA is a general-purpose architecture designed to provide a common technical framework to facilitate the reuse and interoperability of all types of software models and simulations. A simulation or set of simulations developed for one purpose can be applied to a different application under the HLA concept of the federation: a composable set of interacting simulations. While the HLA is the architecture, the RTI software is the actual implementation of federate services to coor-

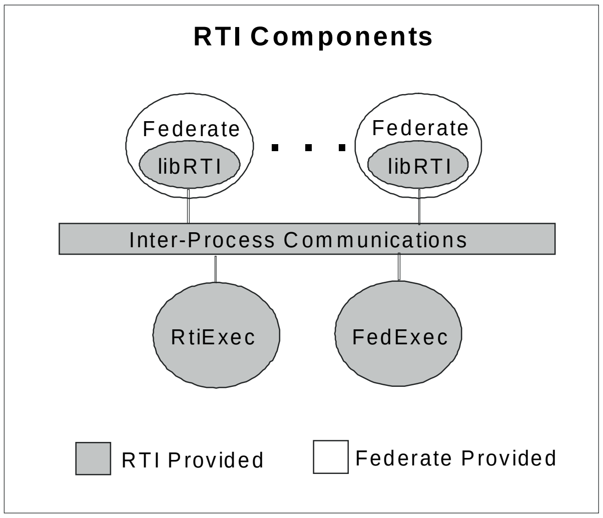

Figure 2. The HLA services are performed via communication between the 2 RTI processes, RtiExec and FedExec, and the federates (simulation programs).

dinate operations and data exchange during a runtime execution.

RTI is a distributed system (Figure 2) that includes two global processes, the RTI Executive (RtiExec) and the Federation Executive (FedExec). The RtiExec's primary purpose is to manage the creation and destruction of federation executions. A FedExec is created for each federation to manage the joining and resigning of federates and perform distribution of reliable updates, interactions, and all RTI internal control messages. The Library (libRTI) implements the HLA API used by C++ programs, and is linked into each federate, with each federate potentially executing on a separate hardware platform.

3.2. RTI testing approach

A typical RTI function performs some type of data management operation involving either an object, ownership of that object, distribution of an object, a declaration, time management, or management of the federation itself. These function calls typically use complex structures as parameters, making testing the RTI functions more complex than simple operating system calls. Testing the RTI function calls involves creating a very simple application composed of a federation containing only one federate that is linked, along with the RTI libraries, to the Ballista testing client. But, setting up even this relatively simple system required creating a federation, creating a federate, having the federate join the federation, and so on. In fact, for every test run on every RTI function call, it was necessary to go through the following nine steps:

1. Ensure that the RTI server (RtiExec) was running
2. Create a federation: registers task with the RtiExec and starts up the FedExec process
3. Join the federation (testing task is a federate)
4. Perform "scaffolding" setup functions
5. Test the actual function
6. Free any memory allocated by the setup functions
7. Resign from the joined federation
8. Destroy federation to de-register from the RtiExec
9. Shut down the RtiExec if last test or error occurred

3.3. Evaluating test results

Ballista tests robustness under exceptional conditions, and is not concerned with whether the result of a function is "correct" (except insofar as the result should be a graceful response to an exceptional situation).

As part of the adaptations for testing the RTI, the previously used "CRASH" scale [15] had to be modified to account for the fact that the RTI API uses exceptions instead of error return codes, and that the RTI has an internal exception handler that attempts to catch hardware-generated signals and perform a "clean" shutdown rather than a raw core dump. The results for RTI testing fall into the following categories, loosely ranked from best to worst in terms of robustness:

- **Pass** - The function call executed and returned normally, with no indication of error.
- **Pass with Exception** - A valid, HLA-defined exception was thrown, indicating a gracefully caught and handled exceptional condition.
- **RTI Internal Error** - RTI encountered an error condition that was not supposed to be possible. RTI did, however, manage to free memory and resign from the federation cleanly. This is the result of a successfully caught hardware memory protection violation, but is not robust per the HLA specification.
- **Unknown Exception** - An unknown exception was thrown and caught internally to the RTI by a catch-all condition (as opposed to a hardware signal). It behaved similarly to the RTI Internal error, but was software-created instead of hardware-triggered.
- **Abort** - An error occurred that was not caught and the code exited immediately (a "core dump"). No memory cleanup, resigning or destroying of the federation took place, requiring a manual restart of the entire federation to resume operation.
- **Restart** - The function call did not return after an ample period of time (a "hang").
- **Catastrophic** - The system was left in a state requiring rebooting the operating system to resume testing.

Based on the above categories, a *robustness failure* was defined for RTI to be any result to executing a test case other than a "Pass" or "Pass with Exception."

4. Testing results

Testing was performed on three different RTI versions:

- Version 1.0.3 (an early version) for Digital Unix 4.0 on an Alphastation 21164
- Version 1.3.5 (current version) for Digital Unix 4.0 on an Alphastation 21164
- Version 1.3.5 (current version) for SunOS 5.6 on a Sparc Ultra-30

Overall a total of 77,338 data points was collected. This number depends on several factors: 1) the number of functions to be tested, 2) the number of parameters in each function, 3) the data types of the arguments, and 4) sampling for functions with very large test sets. The RTI developers made significant changes between versions 1.0.3 and 1.3.5, including adding functions and changing the arguments of existing functions.

Table 1: Directly measured robustness failure rates.

	Functions tested	Functions with RTI Internal Error Exception Failures	Functions with Unknown Exception failures	Functions with Abort failures	Functions with no failures	Number of tests run	Number of RTI Internal Error Exception failures	Number of Unknown Exception failures	Number of Abort failures	Normalized (sum all aborts) rate
RTI 1.0.3 Dunix	76	19	18	0	41	40291	136	2611	0	6.41%
RTI 1.3.5 Dunix	86	20	32	0	43	22757	1255	1965	0	10.20%
RTI 1.3.5 SunOS	86	0	0	45	41	14291	0	0	2289	10.05%

4.1. Normalized failure rates

Table 1 reports directly measured robustness failure rates. In order to avoid problems stemming from different numbers of functions and test cases, a normalized failure rate metric is reported in the last column of Table 1. This failure rate is the arithmetic mean of individual failure rates for each function across all functions tested for a particular RTI/OS combination [15].

In the absence of a particular workload, and given our experience that weighted failure rates vary dramatically depending on the workload (but were in some cases as high as 29% for the POSIX API), it is inappropriate for us to simply take an arbitrary application operating profile and use it here. However, as a simple common-sense check on these results, functions with high robustness failure rates do in fact include commonly used features such as registering an object, publishing data, subscribing to data, and determining attribute ownership.

It is common in software reliability work to use an operational profile for weighting the severity of various problems encountered according to the expected execution frequency of functions (e.g., [17]). Unfortunately, for the RTI, and indeed many general-purpose APIs, this type of profiling data is highly dependent which federation program(s) are running. While we did not have access to realistic RTI programs because that environment itself is still new, data on previous operating system testing showed that operational profile weightings still resulted in significant (10% or more) weighted robustness failure rates [14]. Additionally there is the

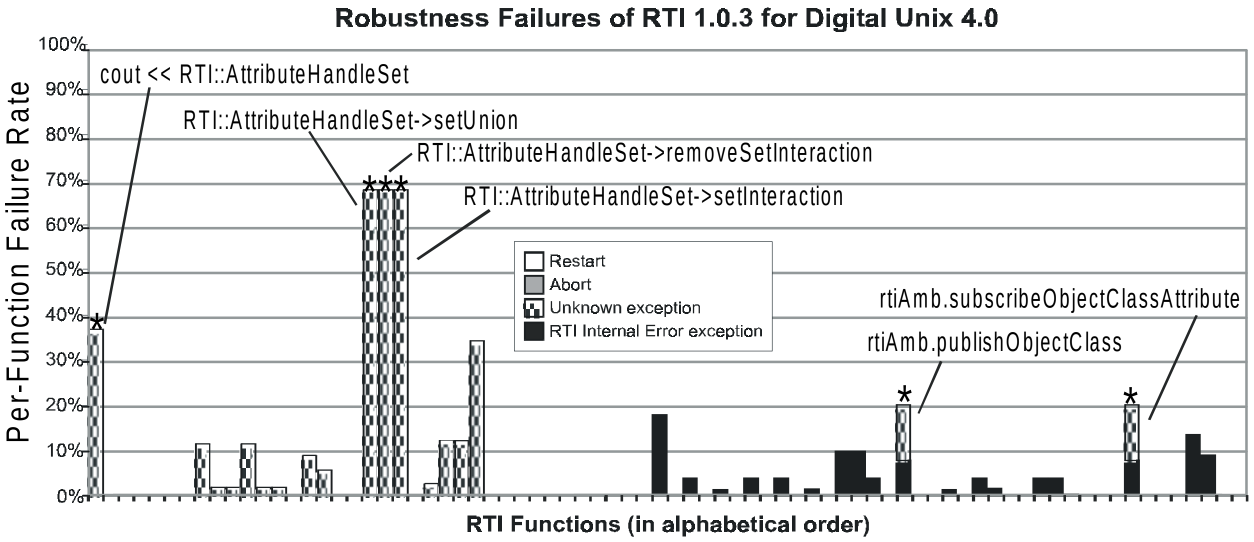

Figure 3. RTI 1.0.3 experienced failures of types RTI Internal Error and Unknown exceptions. Functions marked with "*" all took an RTI::AttributeHandleSet class as a parameter and all had failure rates greater than 20%.

issue that an operational profile for a normally running program is not necessarily applicable to exceptional situations (which are, by definition, abnormal). Thus we do not give detailed weighted failure rate results here, because to do so would risk inviting unwarranted, generalized conclusions by readers drawn from what would be very specific data.

4.2. RTI 1.0.3 for Digital Unix

The types of robustness failures that were detected in RTI version 1.0.3 were RTI Internal Error and Unknown exceptions. Some of the RTI service functions have the ability to throw the exception "`RTIinternalError:Caught unknown exception`." However, in speaking with one of the developers, we learned that this is not supposed to ever occur. A more specific exception should have been thrown instead, rather than making the "RTIinternalError" exception serve as a "catch all" condition or default handler. This type of failure accounted for a 1.4% normalized failure rate, while the Unknown exceptions accounted for a 5.0% normalized failure rate.

As can be seen from Figure 3, the three functions `RTI::AttributeHandleSet->setUnion`, `RTI::AttributeHandleSet->removeSetIntersection`, and `RTI::AttributeHandleSet->setIntersection` responded the least robustly to our tests. All three of these functions took as their sole parameter an `RTI::AttributeHandleSet` class, and all three failed on exactly the same input parameters. In fact, all but one RTI 1.0.3 function we tested that had a failure rate of more than 20% took an `RTI::AttributeHandleSet` class as a parameter (labeled with "*" in Figure 3). The lower failure rates of the two functions with a "*" at 20% were due to masking by a successful second exceptional parameter check before the `RTI::AttributeHandleSet` parameter was touched by the function.

An additional problem discovered while testing was the RTI client process crashing through an RTI service function call. This would occur randomly and the direct cause was never determined. While running a simulation, the following error would occur "`RTIinternalError: Invalid mutex object in RTIlocker::RTIlocker 14001`" for any RTI service function call made. Once in this error state it was impossible to run a federation execution until the machine was rebooted. This error is particularly nasty because it not only forces the user to quit the currently running federation execution without performing any memory clean up or shutdown code, but also requires rebooting the machine before any other RTI function can be executed. This particular problem was not encountered in the two RTI 1.3.5 versions.

4.3. RTI 1.3.5 for Digital Unix and SunOS

The types of robustness failures detected in the two RTI 1.3.5 versions were quite different in manifestation, but similar in profile (Figures 4 and 5). For the robustness failures that were detected in the Digital Unix port, RTI Internal Errors accounted for a 2.6% normalized robustness failure rate, Unknown excep-

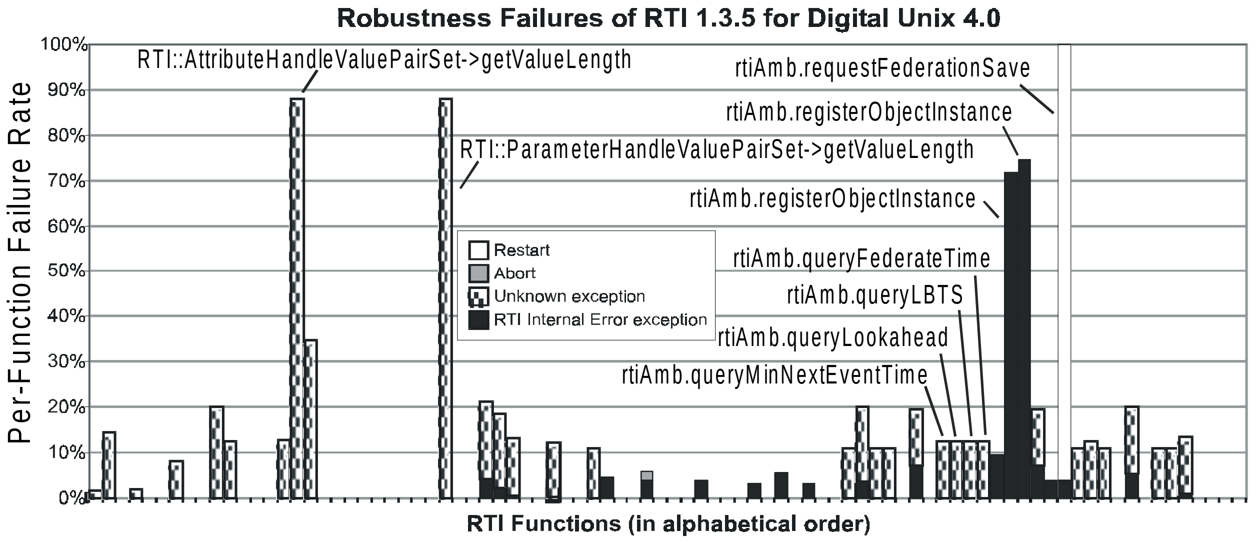

Figure 4. In addition to RTI Internal Error and Unknown exceptions, RTI 1.3.5 for Digital Unix also had one function that experienced multiple restarts, and one function trigger the "stuck in infinite loop" error message.

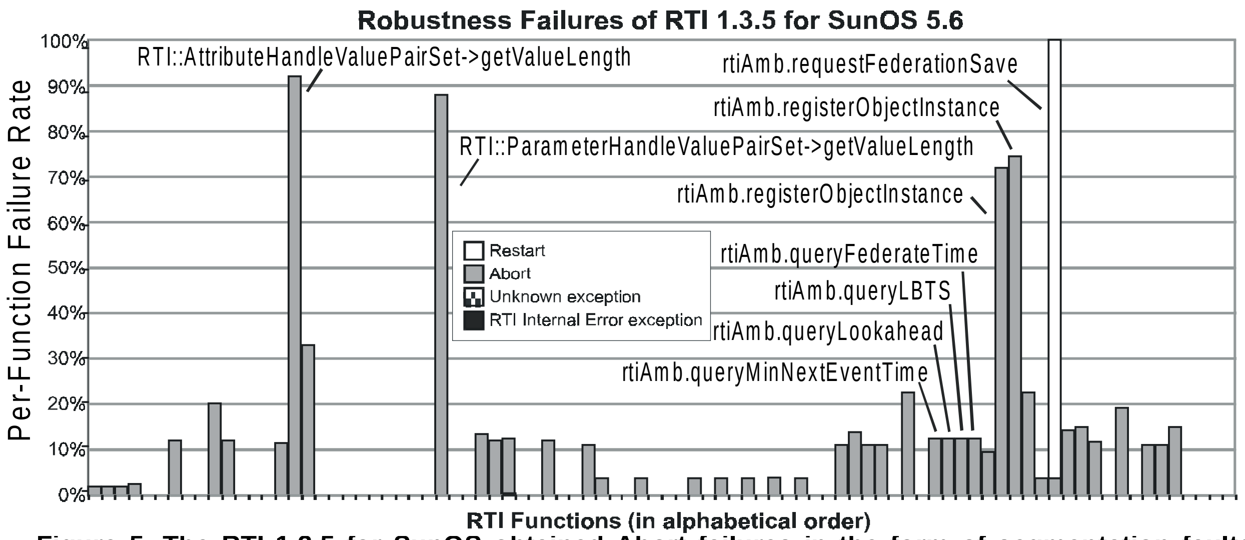

Figure 5. The RTI 1.3.5 for SunOS obtained Abort failures in the form of segmentation faults instead of RTI Internal Errors or Unknown exceptions.

tions accounted for a 6.5% normalized robustness failure rate, Restarts produced a 1.1% normalized robustness failure rate, and one test (counted as an Abort), produced an exception system infinite loop failure which died after printed out the following message:

“Exception system exiting dues[sic] to multiple internal errors:
exception dispatch or unwind stuck in infinite loop
exception dispatch or unwind stuck in infinite loop”.

In comparison, the robustness failures seen in RTI 1.3.5 for SunOS did not include RTI Internal Errors or Unknown exceptions. Instead, two different reactions to exceptional inputs were seen. The first was a segmentation fault that would cause the federate process to exit immediately without properly resigning from and destroying the federation and cleaning up memory. This would result in a “zombie” federate registered with the federation executive. The presence of such zombies caused the subsequently joining federate (the next automatic test case in our situation) to hang in the `rtiAmb.joinFederationExecution` service. To remedy this it was necessary to manually resign from the federation by killing the FedExec and RtiExec processes. The other reaction to an exceptional input that was observed in the SunOS testing was similar to that segmentation fault, except instead of printing out “Segmentation fault” the following message would be displayed followed by execution termination:

“Run-time exception error; current exception: RTI internal error Unexpected exception thrown.”

which appears to indicate an incomplete implementation of an RTI Internal Error. Both of these errors were considered to be Aborts, and accounted 8.9 percentage points of the normalized robustness failure rate. Restart failures accounted for a 1.1 percentage points.

4.3.1. Segmentations faults vs. RTI Internal Error Exception. Comparing the two RTI 1.3.5 graphs shows that the robustness failure rates are essentially the same. However, in the SunOS port, unanticipated signals leak through and are seen as segmentation faults instead of being caught as RTI Internal Errors. The RTI Internal Error seen in the Digital Unix version allows recovery and cleanup, unlike a raw segmentation fault, which aborts the code. The SunOS version’s inability to catch and handle segmentation faults could significantly disrupt the currently running federation execution by failing to inform federates that a task has dropped out. This example serves to illustrate possible problems in porting robust applications across platforms with different exception handling support.

4.3.2. Restart failures. In both implementations, the Restart failures all occurred for the single-parameter function: `rtiAmb.requestFederationSave`. On both Digital Unix and SunOS, 50 of 52 tests hung. As an experiment, we let the `rtiAmb.requestFederationSave` function run for 8 hours, but it remained hung. It is interesting to note that both the two-parameter `rtiAmb.requestFederationSave` function of RTI 1.3.5 and the `rtiAmb.requestFederationSave` function in RTI version 1.0.3 did not have any Restarts, although there were significant changes in this function from version 1.0.3 to version 1.3.5.

4.4. Comparison to operating system results

The results obtained from RTI testing are much more robust than those we obtained testing POSIX operating systems (OSs), which typically had a robustness failure rate between 10.0% to 22.7% [14], compared to the RTI implementations which got between 6.4% and 10.2%. Several of the OSs had catastrophic errors occur, which are failures that occur when the entire OS becomes corrupted or the machine crashes or reboots. In addition, almost every OS of the 15 tested encountered several functions that had Restart failures, whereas only one function in RTI 1.3.5 had Restart failures. We anticipated the results of Ballista testing of the RTI would have a lower failure rate than previous testing of operating systems primarily because the RTI, as well as the HLA, were specifically designed for robust operation.

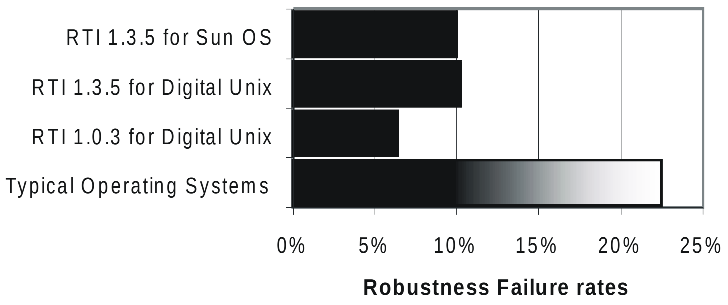

Figure 6. Overall comparison of failure rates of three implementations of the RTI to operating systems (OS failure rates ranged from 10.0% to 22.7%).

We have seen from our previous results of Ballista testing that a newer software version does not necessarily indicate increased robustness. The same holds true for the RTI as illustrated by the two Digital Unix versions' overall robustness ratings (Figure 6). Although many of the RTI 1.3.5 failures may be due to the fact that the RTI specification changed significantly in going to the newer version, it is still interesting to note that based on the normalized robustness failure rates, the newer RTI version, 1.3.5, actually is shown to be less robust. This is not inconsistent with trends seen previously in operating system robustness test results for both major and minor upgrades.

5. Conclusions & future directions

This paper provides the results of Ballista robustness testing of the High Level Architecture Run-Time Infrastructure (HLA RTI), a general-purpose distributed simulation backplane which was specifically designed for robust exception handling. Testing the RTI required significant extensions of Ballista capabilities that were thought by some to be improbable to accomplish without a complete architectural change, including handling exception-based error reporting models, testing object-oriented software structures (including callbacks), incorporating necessary state-setting "scaffolding" code in a scalable manner, and operating in a state-rich distributed system environment. Moreover, these extensions were accommodated through small, natural evolutions of the basic Ballista architecture. This bodes well for extending Ballista to still other application areas according to the project goal of creating a general-purpose, scalable testing framework.

Robustness testing was performed on three different versions of the RTI, with a total of 77,338 data points collected. With a 6.4% to 10.2% normalized robustness failure rate, RTI appears to be significantly more robust than off-the-shelf POSIX operating systems, which had 10% to 22.7% normalized failure rates. As with operating system testing results, certain types of functions were robustness "bottlenecks," having significantly higher failure rates than most other functions. Thus, these testing results should aid in deciding how to allocate developer resources to improve robustness.

The particular robustness problems observed in three version/platform RTI pairs were internal exception handling errors (actually, a semi-gracefully caught segmentation violation) ranging from a 1.4% to a 2.6% normalized failure rate, unknown exceptions (an exception handling software defect) with 5.0% to 6.5% normalized failure rates, and segmentation faults (exceptions that evaded the exception handlers) found only on the SunOS port, accounting for an 8.9% normalized failure rate. Additionally, the Digital Unix port of RTI 1.3.5 suffered "multiple internal errors" on one particular function that required killing the testing task. Finally, the Digital Unix port of RTI 1.0.3 could fail in a way that required rebooting the system to correct. All problems except for the RTI 1.0.3 reboot issue and the one "multiple internal errors" result were readily reproducible and were automatically reduced to simple "bug report" programs by the Ballista server. The code from these bug reports has been added to the RTI developers' regression test suite.

These results indicate that it can be a difficult task to create "bullet-proof" code, even when that is a specifically stated development goal. Additionally, the problem with the SunOS port not catching segmentation faults indicates that it can be difficult to provide comparable exception handling capabilities for the same API across multiple platforms. One piece of good news, however, is that (except for the SunOS problem just mentioned), we did not find significant differences in exception handling coverage across platforms. This suggests, but certainly does not prove, that underlying variations in operating system robustness might not percolate up through well-written exception handling facilities to cause exception handling differences across platforms. If that were to happen, it would further complicate the task of writing portable, robust applications.

In the future, we are working to make Ballista part of the standard verification suite for RTI development. Additionally, we plan to explore issues of concurrent testing to find potentially more subtle bugs related to timing and resource sharing. However, even with a relatively straightforward static, single-thread execution model, Ballista testing has been demonstrated to find exception handling problems in software specifically written to be highly robust.

6. Acknowledgements

This research was sponsored by DARPA contract DABT63-96-C-0064.

7. References

[1] Barton, J., Czeck, E., Segall, Z., Siewiorek, D., “Fault injection experiments using FIAT,” *IEEE Trans. on Computers*, 39(4): 575-82, April 1990.

[2] Carette, G., “CRASHME: Random input testing,” (no formal publication available) http://people.delphi.com/gjc/crashme.html.

[3] Cristian, Flaviu, “Exception Handling and Tolerance of Software Faults,” In: *Software Fault Tolerance*, Michael R. Lyu (Ed.). Chichester: Wiley, 1995. pp. 81-107, Ch. 4.

[4] Defense Modeling and Simulation Office (DMSO). *HLA Homepage General Information*. http://hla.dmso.mil/hla/general/, accessed 4/1/99.

[5] U.S. Department of Defense, High Level Architecture, *Interface Specification version 1.3*, April 2, 1998. Available at http://hla.dmso.mil/hla/tech/ifspec/ifspec-d01-body.pdf.

[6] U.S. Department of Defense, High Level Architecture, *Interface Specification version 1.1*, Feb. 12, 1997. Available at http://hla.dmso.mil/hla/tech/ifspec/ifsp11.pdf.

[7] Department of Defense, High Level Architecture, Run-Time Infrastructure, *Programmer’s Guide*, RTI 1.3 Version 5, Dec. 16, 1998, DMSO, MITRE, SAIC, Virtual Technology Corporation.

[8] Department of Defense, High Level Architecture, Run-Time Infrastructure, *Programmer’s Guide*, RTI 1.0 Version 3, Nov. 14, 1997, DMSO, MITRE, SAIC, Virtual Technology Corporation.

[9] High Level Architecture, RTI 1.3 Version 5, *RTIambassador [API] Reference Manual*, Dec. 16, 1998. Available at http://www.dmso.mil/cgi-bin/hla_dev/hla_cat.pl.

[10] High Level Architecture, RTI 1.3 Version 5, *Supporting Classes Reference Manual*, Dec. 16, 1998. Available at http://www.dmso.mil/cgi-bin/hla_dev/hla_cat.pl.

[11] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990, IEEE Computer Soc., Dec. 10, 1990.

[12] *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Part 1: system Application Program Interface (API) Amendment 1: Realtime Extension [C language]*, IEEE Std 1003.1b-1993, IEEE Computer Soc., 1994.

[13] Kanawati, G., Kanawati, N. & Abraham, J., “FERRARI: a tool for the validation of system dependability properties,” *1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*. Amherst, MA, USA, July 1992, pp. 336-344.

[14] Koopman, P. & DeVale, J., “The Exception Handling Effectiveness of POSIX Operating Systems,” submitted to *IEEE Transactions on Software Engineering*.

[15] Kropp, N., Koopman, P. & Siewiorek, D., “Automated Robustness Testing of Off-the-Shelf Software Components,” *28th Fault Tolerant Computing Symposium*, pp. 230-239, June 23-25, 1998.

[16] Miller, B., Koski, D., Lee, C., Magnanty, V., Murthy, R., Natarajan, A. & Steidl, J., *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*, Computer Science Tech. Report 1268, Univ. of Wisconsin-Madison, May 1998.

[17] Musa, J., Fuoco, G., Irving, N. & Kropfl, D., Juhlin, B., “The Operational Profile”, in: Lyu, M.(ed.), *Handbook of Software Reliability Engineering*, McGraw-Hill/IEEE Computer Society Press, Los Alamitos CA, 1996, pp. 167-216.

[18] Numega. BoundsChecker. http://www.nemega.com/products/vc/vc.html.

[19] Parasoft. Insure++. http://www.parasoft.com/insure/index.html.

[20] Pure Atria. Purify. http://www.pureatria.com/products/purify/index.html.

[21] Tsai, T., & R. Iyer, “Measuring Fault Tolerance with the FTAPE Fault Injection Tool,” *Proceedings eighth International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Heidelberg, Germany, Sept. 20-22 1995, Springer-Verlag, pp. 26-40.

[22] Devale, J. & Koopman, P., "Comparing the Robustness of POSIX Operating Systems," *Fault Tolerant Computing Symposium (FTCS-29)*, pp. 30-37, June 1999.