The Ballista Software Robustness Testing Service

John P. DeVale
Department of Electrical
and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
+1 412 268 4264
jdevale@ece.cmu.edu

Philip J. Koopman
Institute for Complex
Engineered Systems
Carnegie Mellon University
Pittsburgh, PA 15213
+1 412 268 5225
koopman@ece.cmu.edu

David J. Guttendorf
Department of Electrical
and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
+1 412 268 7293
davidg@ece.cmu.edu

ABSTRACT

Computers are becoming essential to everyday life in modern society, but are not necessarily as dependable as one would like, especially with respect to software robustness. Cost and time constraints often limit testing to the important area of functional correctness, but leave few resources to determine a software system's robustness in the face of exceptional conditions. This paper describes the Ballista Project – a methodology and Web server that remotely test software modules in linkable object code form. By focusing on module interfaces rather than functional specifications, we have been able to build a scalable robustness evaluation framework having a test development cost sub-linear with respect to the number of modules to be tested. A client-server framework exercises the module under test on the developer's computer. The Ballista server remotely directs testing, eliminating the need to port server or application code, and maintaining confidentiality of the source code under test. Stand-alone tests to replicate failures are generated and viewed on the World Wide Web. The Ballista server can be used to create or augment test suites, and to collect data to better understand the exceptional conditions that induce software robustness failures.

Keywords

Reliability; Testing, analysis, and verification; Measurement, metrics, empirical methods; Web-based tools, systems, and environments.

1 INTRODUCTION

Although computing systems continue to take on increasingly important roles in everyday life, the dependability of these systems with respect to software robustness may not be as high as one would like. The disruption of communications services or a business server can cause substantial problems for both the service provider and the consumer dependant on the service. Personal computer users have become inured to crashes of popular desktop computer software. But, these same people are notoriously unsympathetic when faced with computer problems that disrupt services in their daily lives.

Preprint: to appear in
Testing Computer Software '99, June1999.

Software robustness failures and exceptional conditions are not a new problem. An early, highly visible, software robustness failure in a critical system came in the Apollo Lunar Landing Program despite the ample resources used for engineering, development and test. In 1969, during powered lunar descent, the Apollo Eleven Lunar Module experienced three computer crashes and reboots due to exceptional sensor and equipment settings.[15]

More recent software development methodologies for critical systems (*e.g.*, [23]) have given engineers the ability to create highly robust systems such as jet engine controllers. But, cost cutting, time pressure, and other real-world constraints on even critical systems can lead to less than perfectly robust software systems. One of the more spectacular recent instances of this was the maiden flight of the Ariane 5 heavy lift rocket. Shortly after liftoff the rocket and payload were lost due to a failure originating from an unhandled exceptional condition during a conversion from a floating point value to an integer value.[19] It stands to reason that everyday projects with lower perceived criticality and budget size are likely to be even more suceptible to robustness problems.

As illustrated by the space flight examples above and most people's daily experience with personal computer software, all too often the response to exceptional conditions is less than graceful. Frequently the focus of software development processes is either that a given input is exceptional and that it should not have been encountered, or that the lack of a specification for that input was a defect in the requirements. But, from a user's perspective, a failure to handle an exceptional condition gracefully amounts to a software failure even if it is not strictly speaking caused by a software defect (it is irrelevant to laypeople that software development documents leave responses to a particular exceptional condition "unspecified" if they have lost anything from an hour's work to a family member due to a software crash).

History and commonsense tell us that specifications are unlikely to address every possible exceptional condition. Consequently, implementations developed by typical software development teams are probably not entirely robust. And, in the world of commercial software, it is even more likely that resource and time constraints will leave gaps where even exceptional conditions that might have been anticipated will be overlooked or left unchecked. As a simple example, consider an ASCII to integer conversion inadvertantly fed a null string pointer expressed by the C call `atoi(NULL)`. As one might expect, on most systems this call causes a segmentation fault and aborts the process. Of course one would not expect programmers to deliberately write "`atoi(NULL)`." But it is possible that a pointer returned from a user input routine purchased as part of a component library could generate a `NULL` pointer value, and that pointer could be passed to `atoi()` during some exceptional condition – perhaps not documented anywhere (just to pick an example, let's say that this happens if the user presses backspace and then carriage return, but that isn't in anyone's test set). Should `atoi()` abort the program in this case? Should the application programmer check the input to `atoi()` even though there is no specified case from the input routine component that could generate a null pointer? Or should `atoi()` react gracefully and generate an error return code so that the application program can check to see if the conversion to integer was performed properly and take corrective action? While it is easy to say such a problem could be handled by a bug patch, a lack of robustness in even so simple a case could cause problems ranging from the expense of distributing the patch to embarassment, loss of customers, or worse.

Resource and time constraints too often allow thorough testing of only the functional aspects of a software system. Functional testing is often the easiest testing expense to justify as well. After all, it is easy to conjure visions of unhappy customers and poor sales to even the most budget-conscious cost cutter if a product fails to perform advertised functions. However, development methodologies, test methodologies, and software metrics typically give short shrift to the issue of robustness. In particular, there has been to date no comprehensive way to quantify robustness, so it is difficult to measure the effects of spending money to improve it. Furthermore, exceptional conditions are typically infrequent, so it is difficult to justify spending resources on dealing with them. (However, it is a mistake to think that *infrequent* or *exceptional* necessarily equates to *unlikely*, as can be demonstrated by year values rolling from 1999 to 2000 with two-digit year data fields; this is infrequent, but certain to happen.)

While robustness may not be an important issue in every software system, those systems which require software robustness can pose special difficulties during testing. In a "robust" software system it is typical for up to 70% of the code to exist for the purpose of dealing with exceptions and exceptional conditions.[10] Unsurprisingly in light of this, other sources state that mature test suites may contain 4 to 5 times more test cases designed to test responses to invalid conditions and inputs ("Dirty" tests) than those designed to test functionality ("Clean" tests).[2] In short, writing robust code and testing for robustness are both likely to be difficult, time consuming, and expensive. Given that the bulk of development effort in such situations is spent on exceptions rather than "normal" functionality, it would seem useful to have tools to support or evaluate the effectiveness of exception handling code.

Other problems arise when attempting to use off-the-shelf software components. It may be difficult to evaluate the robustness of software without access to complete development process documentation (or in some cases lacking even source code!). Yet if robustness matters for an application, it would seem that robustness evaluations would be useful in selecting and using a component library. Evaluations would be even more useful if they were performed in a way that permitted "apples-to-apples" comparisons across the same or similar component libraries from multiple vendors.

The Ballista approach to robustness testing discussed in this paper provides an automated, scaleable testing framework that quantifies the robustness of software modules with respect to their response to exceptional input values. It generates specific test cases that deterministically reproduce individual robustness failures found during testing in order to help developers pin down robustness problems. Additionally, it has been used to compare off-the-shelf software components by measuring the robustness of fifteen different implementations of the POSIX operating system Application Programming Interface (API).

The Ballista methodology uses a portable testing client which can be downloaded and run on a developer's machine along with the module under test. The client connects to the Ballista testing service running on a server at Carnegie Mellon, which directs the client's testing of the module under test. This service allows software modules to be automatically and rapidly tested or characterized for robustness failures with a low cost for test development.

2 PREVIOUS WORK

The Ballista testing framework described here 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 both represent work on automated robustness testing. `Crashme` writes random data values to memory and attempts to execute them as code by spawning a large number of concurrent processes[6]. The Fuzz project injects random noise (or "fuzz") into specific elements of an OS interface[20]. Both methods have found robustness problems in operating systems, although they are not specifically designed for a high degree of repeatability.

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. [16] FTAPE uses hardware-dependent device drivers to inject faults into a system running with a random load generator. [26] These approaches produced useful results, but were not intended to provide a scalable, portable quantification of robustness for software modules.

Typical software testing approaches are only suitable for evaluating robustness when robustness is included as an explicit and detailed requirement reflected in specifications. When comprehensive exceptional condition tests appear in code traceable back to requirements, regular software engineering practices should suffice to ensure robust operation. However, software engineering techniques tend to not yield a way to measure robustness if there is no tracability to specifications. For example, test coverage metrics tend to measure whether code that exists is tested, but may provide no insight as to whether code to test for and handle non-specified exceptions may be missing entirely.

There are tools which test for robustness problems by instrumenting software and monitoring execution (*e.g.*, Purify [23], Insure++ [22], and BoundsChecker [21]). While these tools test for robustness problems that are not necessarily part of the application software specification, they do so in the course of executing tests or user scripts. Thus, they are useful in finding exceptional conditions encountered in testing that might be missed otherwise, but still rely on traditional software testing (presumably traceable back to specifications and acceptance criteria) and do not currently employ additional fault injection approaches. Additionally, they require access to source code, which is not necessarily available. In contrast, the Ballista testing website works by sending selected exceptional inputs directly into software modules at the module testing level, rather than by instrumentating existing tests of an integrated system. Thus it is complementary to, rather than a substitute for, current instrumentation approaches.

3 THE BALLISTA TESTING METHODOLOGY

The Ballista robustness testing methodology is based on combinational tests of valid and invalid parameter values for subroutine calls, methods, and functions. In each *test case*, a single software *M*odule *u*nder *T*est (or *MuT*) is called a single time to determine whether it is robust when called with a particular set of parameter values. These parameter values, or *test values*, are drawn from a pool of normal and exceptional values based on the data type of each argument passed to the MuT. A test case therefore consists of the name of the MuT and a tuple of test values that are passed as parameters (*i.e.*, a test case can in general be described as a tuple: {MuT_name, test_value1, test_value2, ..., test_valueN} corresponding to a procedure call of the form: MuT_name(test_value1, test_value2, ..., test_valueN)) .

Thus, the general approach to Ballista testing is to test the robustness of a single call to a MuT for a single tuple of test values, and then repeat this process for multiple test cases that each have different combinations of valid and invalid test values. A detailed discussion follows.

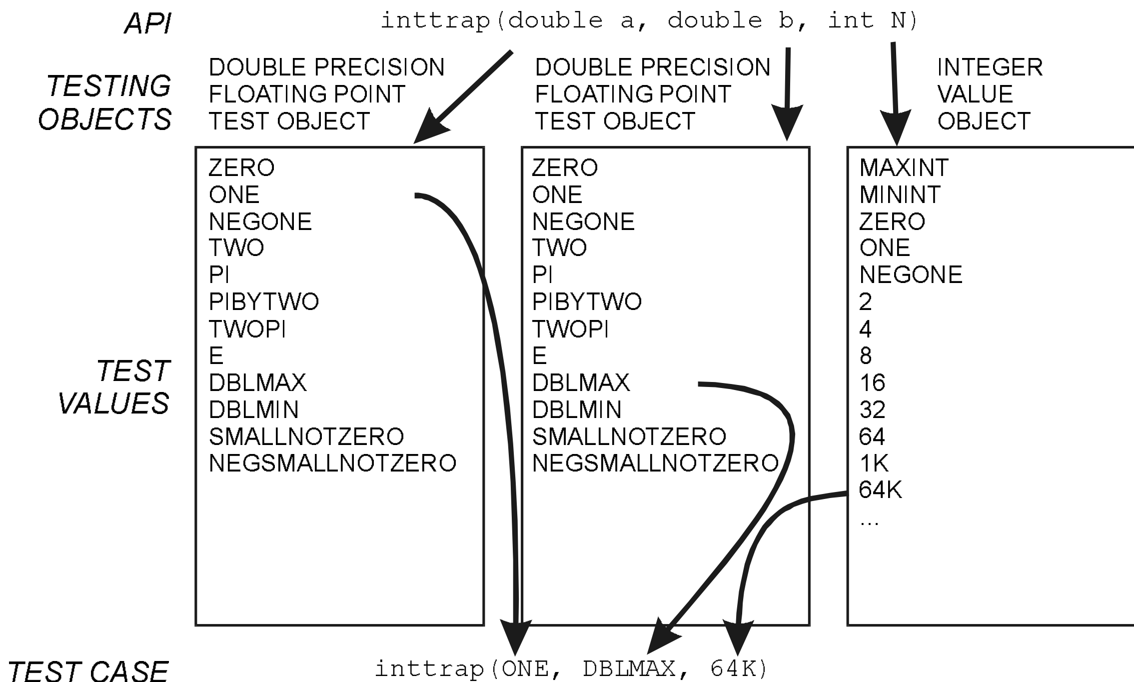


Figure 1. Ballista test case generation for the `inttrap()` function. The arrows show a single test case generated from three particular test values; in general all combinations of test values are tried during the course of testing.

Scalable testing without functional specifications

The Ballista testing framework achieves scalability by using two techniques to abstract away almost all of the functional specification of the MuT. The reasoning is that if the functional specification can be made irrelevant for the purposes of testing, then no effort needs to be made to create such specifications (this is especially important for testing legacy code, code from third party software component vendors, or code with specifications that are not in machine-usable form).

The first technique to attain scalability is that the test specification used by Ballista is simply "doesn't crash; doesn't hang." This simple specification describes a broad category of robust behavior, applies to almost all modules, and can be checked by looking for an appropriate timeout on a watchdog timer and monitoring the status of a task for signs of abnormal termination. Thus, no separate functional specification is required for a module to be tested (those few modules that intentionally terminate abnormally or hang are not appropriate for testing with Ballista).

The second technique to attain scalability is that test cases are based not on the functionality of the MuT, but rather on values that tend to be exceptional for the data types used by the MuT. In other words, the types of the arguments to a module completely determine which test cases will be executed without regard to what that module does. This approach eliminates the need to construct test cases based on functionality. Additionally (and perhaps surprisingly), in full-scale testing it has proven possible to bury most or all test "scaffolding" code into test values associated with data types. This means that to a large degree there is no need to write any test scaffolding code, nor any test cases when testing a new module if the data types for that module have been used previously to test other modules. The net result is that the effort to test a large set of modules in an API tends to grow sublinearly with the number of modules tested. For example, testing 233 functions and system calls in the POSIX API with real time extensions [14] required defining only 20 data types.

For each function tested, an interface description is created with the function name and type information for each argument. The type information is simply the name of a type which Ballista is capable of testing. In some cases specific information about how the argument is used was exploited to result in better testing (for example, a file descriptor might be of type `int`, but has been implemented in Ballista as a specific file descriptor data type).

Ballista bases the test values on the data types of the parameters used in the function interface (Figure 1). For example, if the interface of a function specifies that it is passed an integer, Ballista builds test cases based on what it has been taught about exceptional integer values. In the case of integer, Ballista currently has 22 potentially exceptional test values. These include values such as zero, one, powers of two, and the maximum integer value.

Ballista uses the interface description information to create all possible combinations of parameters to form an exhaustive list of test cases. For instance, suppose a MuT took two floats and a integer as input parameters. Given that there are 22 integer test values and 11 float test values, Ballista would generate 22 x 11 x 11= 2662 test cases.

Implementation of test values

Data types for testing can themselves be thought of as modules, which fit into a hierchical object-oriented class structure. This simplifies the creation of new data types and the refinement of existing types. Each data type is derived from a parent type, and inherits all of the parent's test cases up to the root Ballista type object. For instance, a data type created to specifically represent a date string would have specific test cases associated with it that might include invalid dates, valid dates, dates far in the past, and dates in the future (Figure 2). Assuming the direct parent of the date string data module were string, it would inherit all of the test cases associated with a generic string, such as an empty string, large "garbage" strings, and small "garbage" strings. Finally the generic string might inherit from a generic pointer data type which would be a base type and include a test for a NULL pointer.

Each test value is associated with up to three code segments. The first code segment is a constructor that creates an instance of the data type with specific properties. For example, an integer constructor simply returns an integer value, but a file constructor might create a file, fill it with data, and set certain access permissions. The optional second code segment modifies the effects of the constructor, such as deleting a file while returning a file handle as if the file existed and were open. (This second phase is required to prevent freed resources from being unintentionally recycled if the same data type is used twice in a parameter list.) The third code segment deletes or frees any system resources which may have been allocated by the constructor segments, such as deleting a file or freeing system memory after the test case has been executed.

Date String
Generic String
Generic Pointer
NULL
DELETED
1K
PAGESIZE
MAXSIZE
SIZE1
INVALID
BIGSTRING
STRINGLEN1
ALLASCII
NONPRINTABLE
...
12/1/1899
1/1/1900
2/29/1984
4/31/1998
13/1/1997
12/0/1994
8/31/1992
8/32/1993
12/31/1999
1/1/2000
12/31/2046
1/1/2047
1/1/8000
...

Figure 2. A date string data type inherits test cases from generic string and generic pointer.

Currently, test values are written based on programmer experience and a brief survey of testing literature. Future work will include adding randomized testing and recording fortuitous test values to be reviewed for inclusion the test value database.

Testing results

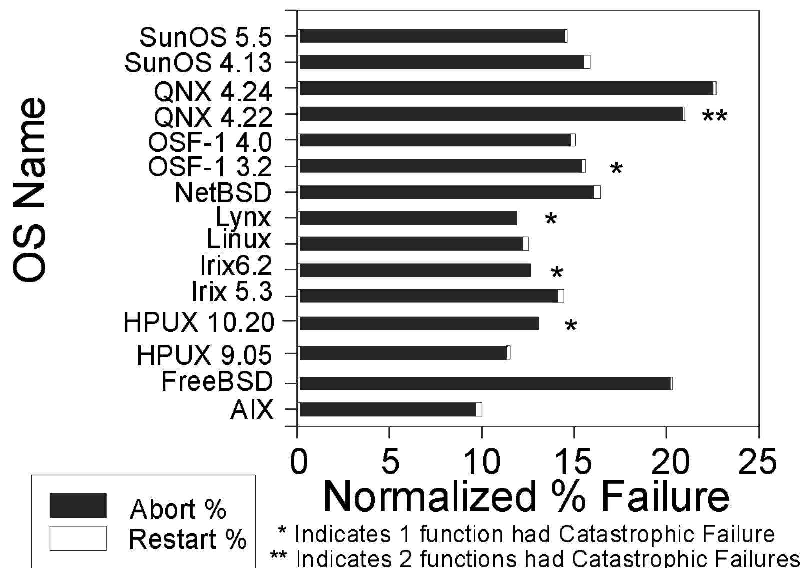
Ballista categorizes test results according to the CRASH severity scale: [18]

- ***C***atastrophic failures corrupt the operating system's state, generally resulting in a machine crash or reboot.
- ***R***estart failures cause the process to "hang" requiring a specific kill signal to be sent to terminate the process after an application-dependent time interval (typically 1 to 10 seconds)
- ***A***bort failures cause abnormal process termination (a "core dump"), and tend to be the most prevalent.
- ***S***ilent failures occur when the MuT returns with no indication of error when asked to perform an operation on an exceptional value. For example, the floating point libraries distributed with several operating systems fail to return an error code, and instead return as if the result were accurate when computing the logarithm of zero[9]. Silent failures can currently be identified only by using inferences from N-version result voting, and so are not identified by the Ballista testing service.
- ***H***indering failures are characterized by the MuT returning an error code that incorrectly describes the exceptional condition that precipitated the error. Hindering failures have been observed to be common in previous work [17], but are not addressed by this version of Ballista due to the lack of a method to automatically identify this class of failure.

Although the Ballista methodology seems unsophisticated, it uncovers a surprising number of robustness failures even in mature software systems. For example, when used to test 233 function calls across 15 different POSIX compliant operating systems, Ballista found normalized robustness failure rates from 9.99% to 22.69%, with a mean of 15.16% (Figure 3)[9]. Preliminary testing of other software systems seems to indicate comparable failure rates.

4 THE BALLISTA WEB SERVER

The next step in Ballista testing has been to create a much more comprehensive testing capability, and has several goals. One goal is to couple testing with data analysis to determine patterns in robustness failures (for example, automated analysis of various tests might reveal that robustness failures are caused by a NULL in the second argument to a module, rather than simply yield a laundry list of robustness failures). A second goal is to create far more comprehensive testing, and couple it with a search engine that identifies "interesting" regions in the test response space to adaptively create tests to perform more sophisticated failure analysis. A third goal is to use the results of testing to create "wrappers" to catch and prevent values from being passed to modules if they are likely to cause robustness failures.



Client-server based remote testing

In order to achieve these future goals and to promote portability, a next-generation Ballista testing system has been written as an Internet-based remote testing service. The Ballista framework is built around a client-server core which allows a minimal client and the MuT to be run on a developer's machine, while the remote Ballista server directs the testing and catalogues the results (Figure 4). The communication between the client and the server is handled by using RPC (Remote Procedure Call [3]) to achieve a high

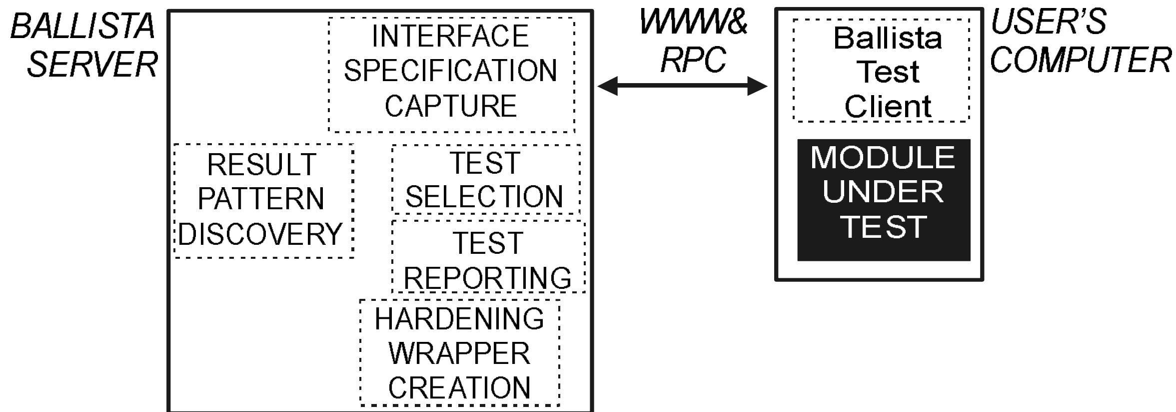


Figure 4. Overview of network test architecture

degree of portability.

The advantages of this client-server architecture go beyond portability (although, for an academic software development group, low-effort portability is a significant benefit to start with!). As Ballista matures in capability these new capabilities can be deployed without going through a software distribution and upgrade cycle at remote sites. With Ballista, these capabilities are expected to not just be software revisions, but also an accumulation of increasingly better heuristics and predefined data types collected from monitoring use of the Web site. Thus, users of the web site will in the normal course of events contribute to increasing its value. Because Ballista testing does not require source code nor specifications (and object code is neither transferred to nor saved at the web site), this data collection can proceed without compromising the confidentiality of proprietary software.

One obvious drawback to this approach is speed. When the client makes an RPC call to the remote server to begin testing, the server generates descriptions of each test value for the entire set of test cases, and transmits them to the client. To generate exhaustive test cases for even a moderately sized input space would require the transmission of several megabytes of data. Experiments on standalone harness testing indicate that random sampling of combinations of test values tends to yield robustness failure rates within one percentage point of actual failure rates. Thus, initially the web site will test a limited number of randomly chosen test value combinations. The second generation of the web site will send batches of tests to reduce startup latency. More sophisticated versions of the web site will use batches of directed, adaptive searching to create more accurate characterisations of robustness failure regions within the module response space. Heuristic techniques from other testing systems such as AETG [5] and TOFU [4] will be adopted as appropriate.

An example of Web Server-based testing

In order to illustrate the usage of Ballista, consider the code segment for performing trapezoidal approximation of numerical integration given in Listing 1. It is typical of textbook algorithms in that is is efficient, and easy to understand. The client-side host was a Digital Unix Alpha Station 500, running Digital Unix 4.0D.

```
double f(double i)
{
  return  i*i*i;
}
double inttrap(double a,double b,int N)
{
   int i; double t=0; double w = (b-a)/N;
  for (i=1;i<=N;i++)
    t+= w*(f(a+(i-1)*w)+f(a+i*w))/2;
  return t;
}
```

Listing 1. Trapezoidal numerical integration algorithm [25] and target function f.

The Ballista testing service web site provides a simple interface with which a user can configure the test harness to run robustness tests on user code. After downloading and uncompressing the client software, all that is required is to simply fill in the appropriate sections of a web form to tell Ballista the name of the module in which the MuT exists, an appropriate header (".h") file so that the C or C++ compiler has function prototype information, the name of the function to be tested, and the appropriate data types for its interface. The Ballista server processes the information supplied by a user (without reading any files from the client disk), and generates a client-side perl script to automatically perform the testing and send test results to the server.

Once testing is complete, the results can be viewed through the web interface. Clicking on the description of a specific test and its result causes the web server to generate a source

program which can be compiled and run on the client-side system. This resulting program can duplicate the result of that test so that failures may be easily reproduced and fixed.

For this example, Ballista ran a total of 2662 tests on the `inttrap()` function in Listing 1. Ballista found 807 tests which caused abort failures and 121 tests which caused restart failures, for a robustness failure rate of 35%. Of course, it is important to realize that any failure rate is going to be directly dependant on the test cases themselves. In this case the MuT was passed two floats and an integer. There were several values for both float and int that were not exceptional for `inttrap()` (such as e and pi). This means that any failure rates are relative, and any comparisons to be made among multiple modules should use the same data type implementations.

On the surface, it seems astonishing that a simple published algorithm such as this would have so many robustness failures. However, in all fairness, this is a textbook algorithm intended to convey how to perform a particular operation, and is presumably not intended to be "bullet-proof" software. Additionally, the Ballista methodology tends to bombard a MuT with many test cases containing exceptional conditions which generate failures having the same root cause. One must examine the details of the test results to obtain more insight into the robustness of the MuT (and it is this process that will be automated in the future).

By looking at the results more closely, it became apparent that all of the restart failures were due to the integer value of MAXINT being used as the value of N, the number of subdivisions. In this case, the algorithm attempted to sum the areas of roughly 2 billion trapezoids. After waiting about 1 second (this value is user configurable) Ballista decided that the process had hung, terminated it, and classified it as a restart failure. While there may be some applications in which this is not a restart failure, in many cases it would be one, and it seems reasonable to flag this situation as a potential problem spot. Thus, the importance of Restart failures found by Ballista varies depending on timing constraints of the application, and will detect true infinite loops if present.

Of the 807 abort failures detected by Ballista for this example, 121 were directly caused by a divide by zero floating point exception. This is probably the easiest type of exceptional condition to anticipate and test for. Seasoned programmers can be expected to recognize this potential problem and put a zero test in the function. Unfortunately, there is a large amount of software developed by relatively unskilled programmers (who may have no formal training in software engineering), and even trained programmers who have little knowledge about how to write robust software.

The remaining 686 aborts were due to overflow/underflow floating point exceptions. This is an insidous robustness failure, in that nearly every mathematical function has the potential to overflow or underflow. This exception is commonly overlooked by programmers, and can be difficult to handle. Nonetheless, such problems have the potential to cause a software system to fail and should be handled appropriately if robust operation is desired.

The floating point class of exceptions can be especially problematic, especially in systems whose microarchitecture by default masks them out. Although this may seem incongruous, the fact that the exception is masked out may lead to problems when hardening algorithms against robustness failures. If the exception does not cause an abort and the software does not check to make certain that an overflow/underflow did not occur, Silent failures (undetectible by the current Ballista service) may result. Resultant calculations may be at best indeterminate and at worst wrong, depending on the application; thus it is strongly recommended that full IEEE floating point features [13] such as propagation of NaN ("Not A Number") values be used even if Ballista testing has been applied to a computational software module.

5 GENERALIZING THE APPROACH

The first-generation, standalone Ballista test harness was developed having in mind POSIX operating system calls as a full-size example system. The results achieved the goals of portability and scalability. However, there were two significant limitations. The POSIX testing harness assumed that any signal ("thrown" exception) was a robustness failure because the POSIX standard considers only error return codes to be robust responses. Second, the result that POSIX testing required absolutely no scaffolding code for creating tests was a bit too good to be true for general software involving distributed computing environment initialization and call-backs. However, both of these issues have been addressed in the Web-based testing system.

Support for exception handling models

The Ballista framework uses ideas on how to build a robust interface developed by the software engineering community. Early robustness work strove to discover multiple ways to handle exceptional conditions. [12] [11] Over the years two methods have come to dominate current implementations. These methods are the termination model and the resumption model . [10]

In current systems the two main exception handling models manifest themselves as error return codes and signals. It has been argued that the termination model is superior to the resumption model. [7] Indeed, the implementation of resumption model semantics in POSIX operating systems (signals), provides only large-grain control of signal handling, typically at the task level resulting in the termination of the process (*e.g.* SIGSEGV). This can make it difficult to diagnose and recover from a problem, and is a concern in real-time systems that cannot afford large-scale disruptions in program execution.

Implementations of the termination model typically require a software module to return an error code (or set an error flag

variable such as `errno` in POSIX) in the event of an exceptional condition. In C++ the use of "thrown" exceptions is also included in the termination model. For instance, a function that includes a division operation might return a divide by zero error code if the divisor were calculated to be zero. The calling program could then determine that an exception occured, what it was, and perhaps determine how to recover from it. POSIX standardizes ways to use error codes, and thus provides portable support for the error return model in building robust systems. [14] Recent work on the Xept method [27] describes a way in which error checking can be encapsulated in a wrapper, reducing flow-of-control disruption and improving modularity.

The Ballista framework supports both the termination model and resumption model as they are implemented in standard C/C++. With the termination model Ballista assumes that exceptions are thrown with some amount of detail via the C++ try-throw-catch mechanism so that corrective action can be taken. In the case of error code returns, it is assumed that there is an error variable that can be checked to determine if an error has occured (currently the `errno` variable from POSIX is supported). Thus, an "unknown" exception or a condition that results in an unhandled generic exception such as a SIGSEGV segmentation violation in a POSIX operating system is considered a robustness failure.

Additionally, Ballista has the ability to add user defined exception handlers to support those modules which use C++ exception handling techniques. Ballista embeds each test call into a standard try-catch pair. Any thrown exception not caught by the user defined catch statements will be handled by the last `catch (...)` in the test harness and treated as an Abort robustness failure.

Support for callbacks and scaffolding

We are in the process of using Ballista to test the High Level Architecture Run Time Infrastructure (HLA RTI -- a simulation backplane system used for distributed military simulations[8]). This example system brought out the need for dealing with exceptions, since errors are handled with a set of thrown exceptions specified by the API. Additionally, the HLA RTI presented problems in that a piece of client software must go through a sequence of events to create and register data structures with a central application server before modules can be called for testing.

While at first it seemed that per-function scaffolding might be required for the HLA RTI, it turned out that there were only 12 equivalence classes of modules with each equivalence class able to share the same scaffolding code. Thus the Ballista Web testing service has the ability to specify scaffolding code (a preamble and a postamble) that can be used with a set of modules to be tested. The preamble also provides a place for putting application-specific "include" files and so on. While there are no doubt some applications where clustering of functions into sets that share scaffolding code is not possible, it seems plausible that this technique will work in many cases, achieving scalability in terms of effort to test a large set of modules.

There are some software systems which require a calling object to be able to support a series of callbacks, either as independant functions or methods associated with the calling object. Two examples of this are a function that analyzes incoming events and dispatches them to registered event handlers, or a registration function which uses member functions of the calling object to obtain the information it requires. These situations require the testing framework itself to be enhanced with either the functions or appropriate object structure to handle these cases.

To facilitate Ballista's ability to test these types of systems we provide a user the ability to build the main testing function call into an arbitrary sub-class, or add any independant functions needed. As with the other customizeable features of Ballista, the additions can be changed between modules. Although this feature adds back some of the per function scaffolding that we eschewed in the previous version of Ballista, it was added to allow testing of code that requires it if desired (and it may be completely ignored for many users).

Phantom parameters – a generalization of the Ballista testing method

At first glance, the parameter-based testing method used by Ballista seems limited in applicability. However, a bit of creativity allows it to generalize to software modules without parameters, modules that take input from files rather than parameters, and tests for how system state affects modules that do not have parameters directly related to that state.

In order to test a module without parameters, all that need be done is create a dummy module that sets appropriate system state with a parameter, then calls the module to be tested. For example, testing a pseudo-random number generator might require setting a starting seed value, then calling a parameterless function that returns a random number in a predefined range. While this could be accomplished by creating preamble test scaffolding to set the starting seed, the Ballista tool supports a more elegant approach. When specifying parameters for a module, "phantom" parameters can be added to the list. These parameters are exercised by Ballista and have constructor/destructor pairs called, but are not actually passed to the MuT. So, for example, testing the random number generator is best done by creating a data type of "random number seed" that sets various values of interest, and then using that data type as a phantom parameter when testing the otherwise parameterless random number generator.

A similar approach can be taken for file-based module inputs rather than parameter-based inputs. A phantom parameter can be defined which creates a particular file or data object (or set of such objects) with a particular format. If that object is accessed other than with a parameter (for example, by referencing a global variable or looking up the object in a

central registry) the phantom parameter can appropriately register the object. In a future version of the Web site, phantom parameters will be able to themselves take parameters, so that for instance a file name could be passed to a generic file creation data type in support of this scheme, and recursive/iterative creation of test sets could be performed.

Given the concept of phantom parameters, it becomes clear how to test modules for system states which are not reflected in explicit parameters. Phantom parameters can be added to set system state as desired. As an example, a phantom parameter data type might be created to fill up disk space before a disk accessing function is called.

Thus, Ballista testing has a reasonable (although certainly not complete) ability to test not only single module calls with single sets of parameters, but the effects of such calls in a wide variety of system states. With the extension of phantom parameters, it can provide highly deterministic testing of what amounts to a wide variety of sequences of function calls (reflected in system state set by that sequence of calls).

6 CONCLUSIONS AND FUTURE WORK

This paper marks the official announcement of public availability of the Ballista robustness testing Web service (at http://www.ices.cmu.edu/ballista). This service can test a large variety of software modules for robustness to exceptional input conditions. Early versions of this testing approach found robustness failure rates ranging from 10% to 23% on a full-scale test of fifteen POSIX operating system implementations. It is expected that applying the testing service to a larger variety of applications will provide valuable insight into both the prevalence of and the types of robustness failures to be found in other varieties of real code.

While at first glance the testing model of a single function call with a single set of parameter seems quite limited, the Ballista testing approach and generalizations of it provide a rich testing capability. Basing tests on parameter types achieves scalability, but still allows a significant amount of system state to be set up while requiring little or no per-function scaffolding. The addition of a "phantom parameter" capability extends the Ballista approach to permit setting up unrelated system state before calling the module under test with a specific set of actual parameter values.

Future work on the Ballista web testing service will include a refined ability to do automatic, fine-grain testing of complex data structures. Additionally, adaptive robustness testing strategies will permit characterizing the regions of input values which evoke robustness failures, and will support an attempt to automatically create software wrappers to detect and gracefully deal with potentially dangerous parameter input values.

7 ACKNOWLEDGEMENTS

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

REFERENCES

[1]Barton, J., Czeck, E., Segall, Z., Siewiorek, D., "Fault injection experiments using FIAT," IEEE Transactions on Computers, 39(4): 575-82.

[2]Beizer, B., *Black Box Testing*, New York: Wiley, 1995.

[3]Bloomer, J. Power Programming with RPC, O'Reilly & Associates

[4]Biyani, R. & P. Santhanam, "TOFU: Test Optimizer for Functional Usage," *Software Engineering Technical Brief*, 2(1), 1997, IBM T.J. Watson Research Center.

[5]Cohen, D., Dalal, S., Fredman, M. & Patton, G. "The AETG System: an approach to testing based on combinatorial design," IEEE Transactions on Software Engineering, vol. 23, no. 7, pp. 437-44.

[6]Carrette, G., "CRASHME: Random input testing," (no formal publication available) http://people.delphi.com/gjc/crashme.html accessed July 6, 1998.

[7]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.

[8]Dahmann, J., Fujimoto, R., & Weatherly, R, "The Department of Defense High Level Architecture," Proceedings of the 1997 Winter Simulation Conference, Winter Conference Board of Directors, San Diego, CA 1997.

[9]DeVale, J & Koopman P., "N-Version approach to Operating System Robustness," 29th Symposium on Fault Tolerant Computing Systems, 1999

[10]Gehani, N., "Exceptional C or C with Exceptions," Software - Practice and Experience, 22(10): 827-48.

[11]Goodenough, J., "Exception handling: issues and a proposed notation," Communications of the ACM, 18(12): 683-696, December 1975.

[12]Hill, I., "Faults in functions, in ALGOL and FORTRAN," The Computer Journal, 14(3): 315-316, August 1971.

[13]IEEE standard for binary floating point arithmetic, IEEE Std 754-1985, Institute of Electrical and Electronics Engineers, 1985.

[14]IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) Amendment 1: Real-time Extension [C Language], IEEE Std 1003.1b-1993, IEEE Computer Society, 1994.

[15]Jones, E., (ed.) The Apollo Lunar Surface Journal, Apollo 11 lunar landing, entries at times 102:38:30, 102:42:22, and 102:42:41, National Aeronautics and Space Administration, Washington, DC, 1996.

[16]Kanawati, G., Kanawati, N. & Abraham, J., "FERRARI: a tool for the validation of system dependability proper-

ties," 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems. Amherst, MA, USA, July 1992, pp. 336-344.

[17]Koopman, P., Sung, J., Dingman, C., Siewiorek, D. & Marz, T., "Comparing Operating Systems Using Robustness Benchmarks," Proceedings Symposium on Reliable and Distributed Systems, Durham, NC, Oct. 22-24 1997, pp. 72-79.

[18]Kropp, N., Koopman, P. & Siewiorek, D., "Automated Robustness Testing of Off-the-Shelf Software Components," 28th Fault Tolerant Computing Symposium, in press, June 23-25, 1998.

[19]Lions, J.L. (chairman) Ariane 5 Flight 501 Failure: report by the inquiry board, European Space Agency, Paris, July 19, 1996.

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

[21]Numega. BoundsChecker. http://www.numega.com/products/vc/vc.html, accessed 8/28/98.

[22]Parasoft. Insure++. http://www.parasoft.com/insure/index.html, accessed 8/28/98.

[23]Pure Atria. Purify. http://www.pureatria.com/products/purify/index.html, accessed 8/28/98.

[24]RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification, Document, SC-167/Eurocae WG-12, RTCA, Washington DC, 1992.

[25]Sedgewick, Robert. *Algorithms in C++*. Addison-Wesley, 1992.

[26]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.

[27]Vo, K-P., Wang, Y-M., Chung, P. & Huang, Y., "Xept: a software instrumentation method for exception handling," The Eighth International Symposium on Software Reliability Engineering, Albuquerque, NM, USA; 2-5 Nov. 1997, pp. 60-69.