Model-Based Statistical Testing of a Cluster Utility

W. Thomas Swain[1] and Stephen L. Scott[2,*]

[1] Software Quality Research Laboratory,
University of Tennessee Department of Computer Science Knoxville, Tennessee 37996
[2] Network and Cluster Computing Group,
Computer Science and Mathematics Division,
Oak Ridge National Laboratory, Oak Ridge, Tennessee
swain@cs.utk.edu, scottsl@ornl.gov

Abstract. As High Performance Computing becomes more collaborative, software certification practices are needed to quantify the credibility of shared applications. To demonstrate quantitative certification testing, Model-Based Statistical Testing (MBST) was applied to *cexec*, a cluster control utility developed in the Network and Cluster Computing Group of Oak Ridge National Laboratory. MBST involves generation of test cases from a usage model. The test results are then analyzed statistically to measure software reliability. The population of *cexec* uses was modeled in terms of input selection choices. The J Usage Model Builder Library (JUMBL) provided the capability to generate test cases directly as Python scripts. Additional Python functions and shell scripts were written to complete a test automation framework. The resulting certification capability employs two large test suites. One consists of "weighted" test cases to provide an intensive fault detection capability, while the other consists of random test cases to provide a statistically meaningful assessment of reliability.

1 Introduction

The work described here had two primary objectives: (1) to certify the *cexec* command in the Cluster Command and Control (C3) tool suite [1,2], and (2) to demonstrate Model-Based Statistical Testing (MBST) as a certification methodology for computational software. Briefly C3 is a set of command line utilities to facilitate management of Linux clusters. The *cexec* command invokes a specified application on any combination of nodes in a Linux cluster.

MBST treats software testing as a statistical experiment. That is, each test is viewed as a sample from the population of all possible uses. This approach to testing involves six tasks: (1) usage model definition, (2) model analysis, (3) test automation, (4) test case generation, (5) test execution, and (6) results analysis

MBST is applied to systems by mapping input stimulus sequences to states of use and associated responses. However computational programs often have only two

* Research supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, Office of Science, U. S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3514, pp. 443 – 450, 2005.
© Springer-Verlag Berlin Heidelberg 2005

states of use: pre-execution and post-execution. Such programs are further characterized by multiple input parameters, often with very large domains.

For *cexec*, usage was modeled in terms of the input selection process. The model analysis, performed using the J Usage Model Builder Library (JUMBL) [3], provided information needed to plan details of the testing strategy. Once a testing strategy was defined, a test automation framework was implemented using Python and *bash* shell scripts. From the model, JUMBL was used to generate test cases. Test cases for all paths through the model were generated in decreasing order of probability; and a set of random test cases were generated to be statistically representative of the usage profile embodied in the model. All test cases were executed using the test automation framework. The pass/fail outcome of all test cases was determined by automated inspection, supplemented by manual analysis of reported failures. The results were then analyzed statistically to provide a quantitative basis for certification.

2 Model Definition

Model definition requires four elements: (1) definition of the test boundary, (2) definition of a *use*, (3) identification of all input stimuli, and (4) identification of correct system responses.

The test boundary is defined by the list of interfaces where stimuli can be applied and responses can be observed. The interface list for *cexec* includes the following:

- The interface by which command line arguments are supplied on invocation.
- The interface by which environment variables are supplied.
- The file system interface by which files are read, written, or deleted.
- The interfaces to `stdout` and `stderr`.
- Other interfaces provided by UNIX system calls.

The definition of a use is simply execution of *cexec* by sending its command line to the operating system of the cluster head node. For each use, the command runs to completion with no intervening stimuli. Consequently sampling the *use* population is a matter of sampling the set of all possible input combinations. Modeling the input selection process as a discrete Markov chain allowed JUMBL to be used for test case generation and test analysis. With this approach, test cases are defined by input combinations obtained using the model as a statistical sampling mechanism.

The set of all possible combinations of the input parameters is quite large, even for a utility such as *cexec*. In addition to some discrete command line options, *cexec* input may include the following inputs as arbitrary character strings: Machine Definition expressions (to specify a subset of the cluster), UNIX command strings, configuration file names, and configuration file contents.

To cope with the large domains of these parameters, the following abstractions were used:

- Machine Definitions = [good, bad, none]
- UNIX commands = [good, bad]
- configuration file names = [good, bad, missing (when expected)]
- configuration files = [good, bad]

For test execution, sampling within these subdomains was built into the test automation framework.

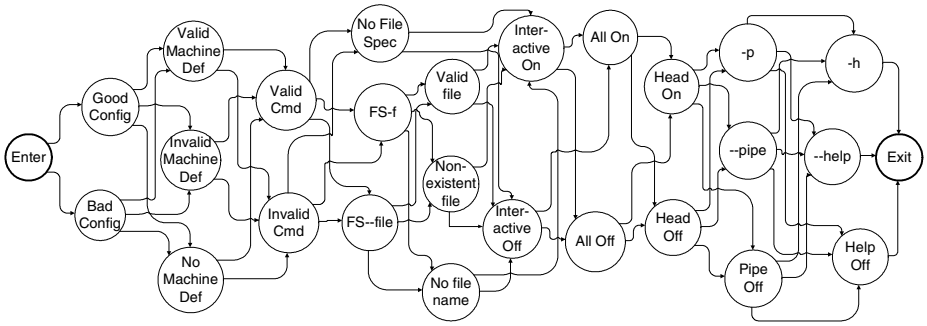

Fig. 1. *cexec* Usage Model

The resulting input selection model is illustrated graphically in Figure 1. The nodes of the graph represent input parameter values (or abstracted ranges), and the arcs represent individual value choices. Probabilities (not shown in the figure) are assigned to the arcs to indicate the relative likelihood of each value for a particular parameter. The probabilities used were based on the following assumptions:

- Except as noted below, all valid choices for a parameter are equally probable.
- Valid choices for a parameter are about 100 times more likely than invalid choices.
- A Machine Definition will be specified for 90% of all commands.
- About half of all commands will specify the configuration file on the command line instead of using the default configuration file.

3 Model Analysis

Analysis of the model as a Markov chain is used for model validation and test planning. Table 1 shows key results from the model analysis.

Table 1. *cexec* Usage Model Analysis

Node Count	27 nodes
Arc Count	60 arcs
Statistically Typical Sequences	$2^{6.65}$ cases
Maximum Mean First Passage	200 cases
Mean First Passage Variance	39,800

The model analysis results can be useful for both model validation and test planning. In this case the model structure is simple enough that the structure and probabilities can be validated by inspection.

For test planning, the analysis provides insights that are not available from inspection of the model. First the number of Statistically Typical Sequences given in Table 1

indicates that $2^{6.65}$ (~100) random test cases are needed to produce a test population that is typical of the usage profile embodied by the model. Additionally the maximum Mean First Passage indicates the average number of random test cases needed to encounter every node of the model is 200 with a variance of 39,800. The corresponding standard deviation σ is on the order of 200. Using the semi-quantitative argument that $>3\sigma$ test cases would give high confidence of complete model coverage, 1000 random test cases were planned.

4 Test Automation

Figure 2 shows an overview of the test automation data flow. The processes in the figure are grouped into the following major automation tasks:

- Test case generation (*cexec* command line construction),
- Generation of configuration files and Machine Definitions,
- Results checking and comparison.

Most of the automation framework is reusable for other C3 commands or future versions of *cexec*. Only the usage model definition file and the Python functions used to check test results would need to be changed.

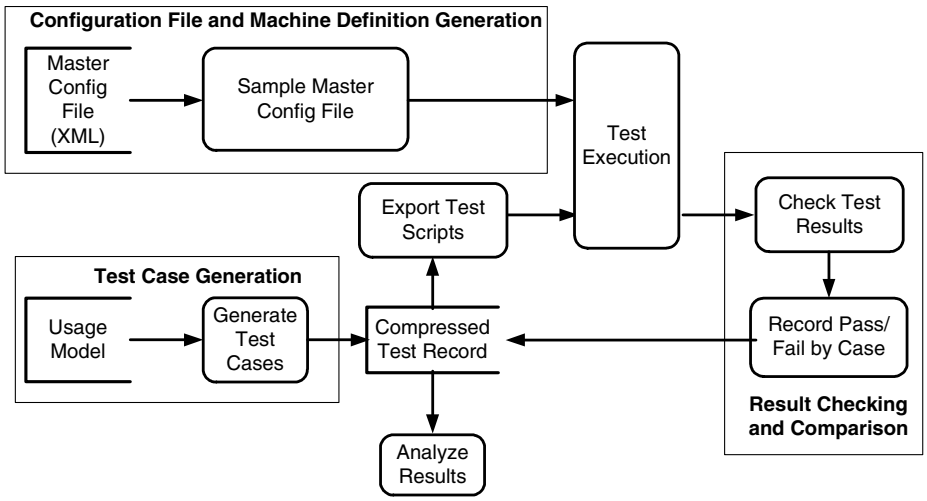

Fig. 2. Test Automation Overview

Automated Command Line Construction

To support test automation, the TML modeling language [4] allows each element (model, state, or arc) to be annotated with arbitrary text, referred to as a "label" in TML syntax. Test cases are initially generated and stored in a compressed Saved Test Record (STR) format. For test execution the JUMBL *managetest export* command [5]

is used to convert STR to a human-readable form. The exported test case is a sequential listing of the labels associated with the model elements encountered.

Labels in the *cexec* TML file consist of Python statements. Initialization code is associated with the `[Start]` state. Python code in arc labels assigns values to *cexec* input parameters. Code associated with state labels records inputs for use in results checking. The label on the `[Exit]` state consists of code required to assemble the *cexec* command line, spawn the resulting command, and record and evaluate the output. Thus each exported test case is a short Python program that automates execution of the required command string and comparison of actual and expected output.

Configuration File and Machine Definition Generation

As mentioned above, four of the *cexec* input parameters are character strings and were each abstracted into two or three discrete values in the usage model. During test case generation, each of these parameters is assigned one of its abstract values. At test case run time, the assigned abstract value must be replaced with a specific value from the associated subdomain. For example, "good" file names are created as `<test case name>.conf`, and "bad" file names are always `bogus.conf` (a non-existent file).

Since configuration files and machine definitions are processed entirely within *cexec*, a much more varied sampling of their parameter spaces is desirable. Figure 3 shows an overview of this sampling process.

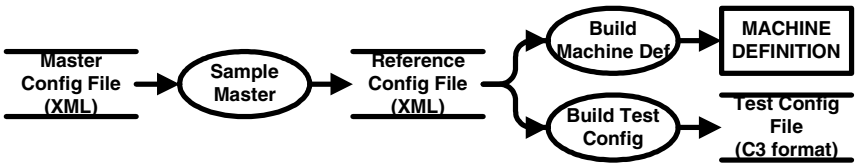

Fig. 3. Data Flow for Generating Configuration Files and Machine Definitions

To expedite data format conversion, an XML format was defined to represent the content of a C3 configuration file. The Master Config File in Figure 3 is the XML representation of the complete test system. At test case run time, a subset of clusters and nodes is selected from the Master Config File and used to create the Reference Config File. The Reference Config File is then transformed to the Test Config File in the standard C3 configuration file format.

If the test case calls for a "bad" configuration file, a single fault is inserted in the file by the Build Test Config process. The type of fault and the insertion location are selected randomly from among the syntactic elements that compose a C3 configuration file. If the test case requires a Machine Definition expression in the command line, the clusters and nodes to be included are selected and composed into the required string by the Build Machine Def process in Figure 3. When required, machine definition fault injection is handled in a manner similar to that for configuration files.

Results Checking

The standard output of *cexec* for each test case is redirected to a log file. The contents of the log file are then compared to the expected output. The graphs in Figures 4 and

5 depict the mapping of input combinations to responses. Each shaded leaf node in the graph represents a set of input combinations for which the expected output can be determined from a single rule.

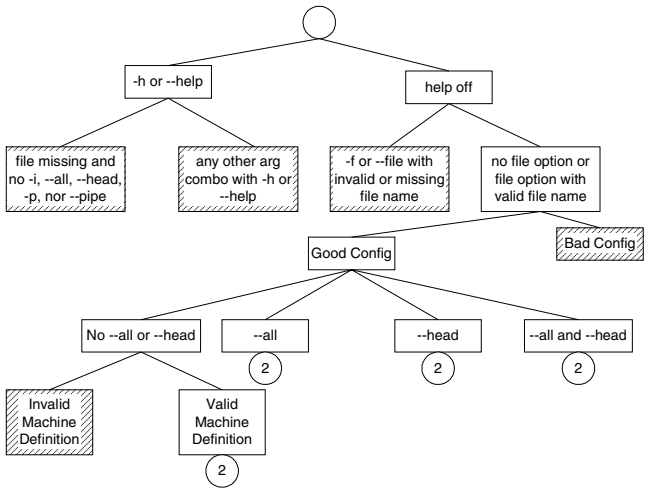

Fig. 4. Top Level *cexec* Response Tree

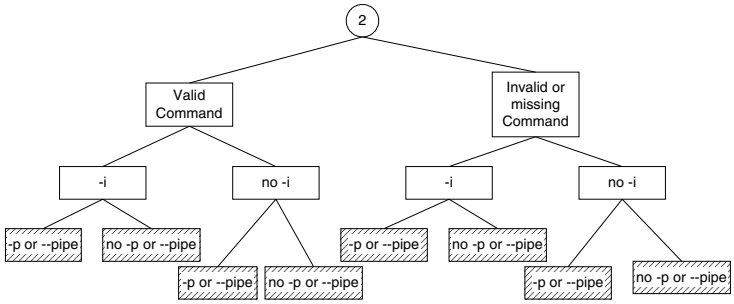

Fig. 5. Continuation of *cexec* Response Tree

Based on these rules for the expected output, a Python function was written to make a run-time pass/fail determination for each test case. For each test case, this function writes the test case name and a summary of the *cexec* input parameters to a file. If a test case failure is detected, additional diagnostic information is written to the file. If *cexec* terminates abnormally, the UNIX exit code is reported. If *cexec* terminates normally, the output line causing the failure determination is reported along with the expected result. Finally the test case is marked as a failure in the STR file.

A few cases involving invalid Machine Definition input were reported as failed by the run-time checking and required manual inspection to screen out the false failures. However these cases were easily recognized in the summary report.

5 Test Case Generation and Execution

The JUMBL provides five test case generation options: state and arc coverage, random, probability-weighted, cost-weighted, and manually crafted. The following factors were considered in selecting the test generation options to be used:

- Since the model involves relatively coarse abstractions for configuration file and Machine Definition inputs, minimal state and arc coverage is not adequate.
- With automation, up to several thousand test cases can be run within reasonable time and cost.
- Since quantitative certification is a primary objective, a subset of the test cases should represent a statistically typical sample (based on the model).
- Based on the model analysis above, the statistically typical sample requires 500 to 1000 random test cases.
- Since the model contains no loops or cycles, it is possible to test all possible paths through the model.

Based on these factors, the probability-weighted generation option was selected as a way to produce test cases for all possible paths through the model. This process generated 6048 test cases. To provide a statistically typical sample, 1000 random test cases were also generated.

The tests were executed on ORNL's XTORC cluster. As described above, each test case was exported as a Python program. One *bash* shell script was used to sequence the weighted test cases, and a second was used for random test cases. Execution of all 7048 test cases takes about four hours.

6 Results Analysis

Table 2 shows a summary of the test results. Sample Reliability is a simple ratio of the number of tests passed to the total for each type (*i.e.*, assumes Bernoulli sampling). Miller reliability is a Bayesian reliability estimate described in [6].

Table 2. *cexec* Certification Test Results

Case Type	Cases	Pass	Fail	Sample Reliability / Variance	Miller Reliability / Variance
Weighted	6048	5837	211	0.965/0.0337	0.965/0.00564
Random	1000	999	1	0.999/0.000999	0.998/0.000332
Combined	7048	6836	212	0.970/0.0292	0.970/0.00488

For weighted test cases, each path through the model is included exactly once in the test. This fact causes low probability test cases to be over-represented in the weighted sample relative to typical usage. The random test case sample, and therefore its reliabilities, reflects more realistic assumptions regarding actual usage.

The total of 212 failures observed among all test cases resulted from six distinct faults. All of the faults involved failure of *cexec* to handle input errors gracefully. The

test results also included two significant observations regarding undocumented behavior of *cexec*: (1) Non-existent nodes in the Machine Definition are ignored, and (2) duplicate nodes in the machine definition are processed as many times as they appear.

7 Conclusions

The purpose of the work described above was twofold: (1) demonstration of MBST for computational software and (2) certification of the C3 *cexec* utility.

Using *cexec* as the system under test, this effort demonstrated that the methods and tools developed for MBST could be applied effectively to computational software. The key adaptation required is to model the input selection process instead of the discrete behavior. Using the test case generation and annotation features of the JUMBL tool set, it was possible to generate automatically a large number of test cases whose execution could be readily automated.

The following are key elements of the test automation framework:

- Direct generation of test cases in the form of Python programs,
- Python classes to generate run-time values for input parameters represented by abstractions in the model,
- Python utilities to capture *cexec* output, compare it to expected results, and record pass/fail information.

With this framework, *cexec* can be thoroughly tested in about four hours, and the automation framework is largely reusable.

In terms of *cexec* certification, the test cases generated using the "weighted" algorithm exercised every input combination at the defined level of abstraction. While the weighted test cases proved quite thorough in detecting even obscure software faults, the random test cases provide the primary basis for certification. The random test cases represent a statistically typical usage sample based on the model. The results of these tests indicate that users can expect the *cexec* program to function properly 99.8% of the time.

References

1. "Project C3 Cluster Command and Control," Network and Cluster Computing Group, Computer Science and Mathematics Division, ORNL, http://www.csm.ornl.gov/torc/C3.
2. M.Brim, R.Flanery, A.Geist, B.Luethke, and S.L.Scott, "Cluster Command and Control (C3) Tool Suite," pp. 381-399, *Parallel and Distributed Computing Practices Special Issue: Quality of Parallel and Distributed Programs and Systems*, Ed: P. Kacsuk and G. Kotsis, Vol. 4, No. 4, December 2001, issn 1097-2803, Nova Science Publishers Inc.
3. S. Prowell, "JUMBL: A Tool for Model-Based Statistical Testing", *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)*, January 2003.
4. S. Prowell, "TML: A description language for Markov chain usage models", *Information and Software Technology*, Vol. 42, No. 12, September 2000, 835--844.
5. JUMBL 4 User's Guide, Software Quality Research Laboratory, Department of Computer Science, University of Tennessee, November 22, 2002.
6. K. Sayre, J. Poore, "A Reliability Estimator for Model Based Testing", *Proceedings of the Thirteenth Symposium on Software Reliability Engineering*, November, 2002.