An Integrated General Purpose Automated Test Environment

Peter A. Vogel
Convex Computer Corporation
P.O. Box 833851
Richardson, TX 75083-3851
pvogel@convex.com

ABSTRACT

As software systems become more and more complex, both the complexity of the testing effort and the cost of maintaining the results of that effort increase proportionately. Most existing test environments lack the power and flexibility needed to adequately test significant software systems. The CONVEX Integrated Test Environment (CITE) is discussed as an answer to the need for a more complete and powerful general purpose automated software test system.

1. INTRODUCTION

It has long been recognized that the potential benefits of automated testing are enormous. Productivity is improved because less time is spent by test engineers performing "monkey" duties such as repetitive execution of tests. Job satisfaction is improved because test engineers can apply themselves to the creative aspects of designing good tests. Product quality is improved because more tests can be written and more tests can be executed within a given schedule, which also improves the product time to market. In addition, automated testing improves the likelihood that results can be reliably reproduced.

Unfortunately, most automated test systems available today [Sit91] are too limited to effectively deliver all of the advantages of automated testing. Although many computer aided software test tools are available today, most are limited to automating only one part of the test effort. For those systems that automate test execution, maintenance of tests can be extremely difficult.

The CONVEX Integrated Test Environment (CITE) was developed to address these problems and others. CITE provides a test environment which is used to test virtually all software products produced by Convex, from system diagnostic software through operating system kernels and utilities, compilers and graphic applications such as AVS. CITE is, as far as we know, the most powerful and complete automated testing environment in use today.

Section 2 presents an overview of CITE's capabilities, focusing on the rule-based approach to test case definition. Section 3 details the rule-based approach to test case definition and execution. Section 4 presents an example use of CITE to test a C compiler. Section 5 discusses the past and present use of CITE at Convex. Section 6 concludes with a brief look at future directions for CITE.

ACM-ISSTA'93-6/93/Cambridge, MA, USA

2. OVERVIEW OF CITE

CITE is made up of many components, each component covers a specific area of automated test management. Figure 1 shows the interaction of the primary CITE components. Some components, such as the test execution system, were developed entirely at Convex, others, such as the *GCT* test coverage tool[Mar93] and the *Expect* interactive session simulator[Lib92] are based on utilities obtained from the public domain. Figure 1 shows only utilities that were developed at Convex. Note that the test processes invoked by the test driver may be invocations of other CITE tools, such as the X window testing utilities described in [Min92].

2.1 TEST EXECUTION

The focus of CITE is the execution of test suites, which are hierarchical collections of test cases. At the leaves of a test suite hierarchy are *basesuites*, which contain a database of tests and the files used by the tests.

Each test can be defined to run as a specific user, and/or to run with a specific primary group. This ability permits tests to be written which exercise utilities that can only be run by the superuser, as well as allowing tests to be written which modify user configuration files without affecting real users of a system.

The test driver, *td*, copies all files used by a test to an execution directory, which becomes the current directory for the test to be executed. The environment is initialized according to information in the test database and the test is executed as a child process of the test driver. The results of each test process are verified against the expected results for the test stored in the database. The test driver maintains a report about the execution of test cases, including the configuration of the system at the time tests began execution and the results of each test. by default, the files used and created by a test are kept in the execution directory if a test fails. The report and the execution directory can later be used for failure analysis.

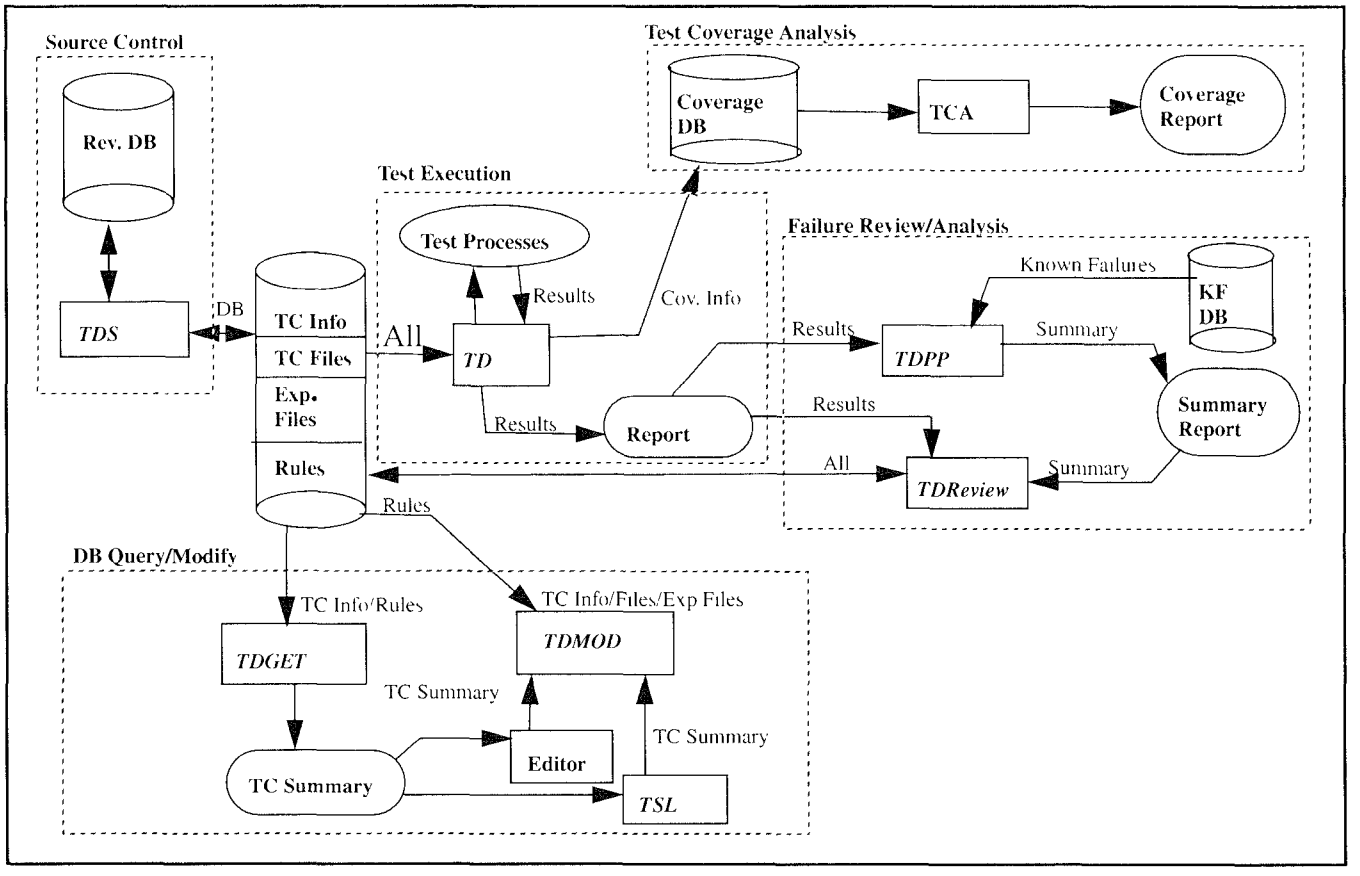

FIGURE 1: INTERACTION OF CITE COMPONENTS

2.1.1 TEST LISTS

In addition to the standard hierarchical organization of tests, within each basesuite tests can be arbitrarily grouped into *test lists*. The these lists can be combined using standard set operations (union, intersection and complement) to specify a specific set of tests to be operated on by the test utilities.

With test lists, a single hierarchical collection of tests can have several "virtual" organizations for different purposes. For example, the compiler test suites define lists of tests that are known to produce a high level of coverage for specific areas of the compiler in a short period of time. These lists are used to quickly verify changes to the compiler before checking them in.

In addition to the user-defined lists, the test driver will, by default, not run tests in the FAIL list in order to prevent the waste of valuable machine resources on tests known to fail. Further, if a test in the list named CRITICAL fails, no further tests in the basesuite will be executed. This facility is used to prevent execution of tests that will fail if some other prior test failed to set up a proper environment, or to prevent resource waste running complex tests against functionality that fails when tested in a very simple manner.

2.2 TEST COVERAGE ANALYSIS

While the execution of test cases is the primary element of automated testing, it is desirable to know, quantitatively, how much of a product's code was executed by the execution of a set of tests. For this purpose, CITE provides two tools to produce a coverage analysis of test execution.

2.2.1 STATEMENT LEVEL (BLOCK) ANALYSIS

The most basic coverage information is a report of what statements of a product's code were or were not executed by the execution of some test set. CITE supplies the *tca* tool which makes use of the profiling data written by a program compiled for basic-block profiling to determine the code coverage. This information can then be used to improve a test suite.

2.2.2 DETAILED COVERAGE ANALYSIS

In [Mat91], it is reported that over 90% of faults in product code can be detected with a test set that satisfies the weakest coverage criteria, that is, block-level coverage. This may be sufficient, however, there are instances where more detailed coverage analysis is desired. For example, significant areas of new code in a critical component of a production software product may require

detailed coverage analysis.

For more detailed analysis, CITE includes the public-domain GCT coverage tool, a modified GNU C compiler, which provides data regarding branch (decision) and loop coverage, as well as operator and operand mutation coverage. [Mar93]

2.3 FAILURE REVIEW AND ANALYSIS

Just as products may have defects, tests themselves may have defects which result in a false failure. While the test definition and execution mechanism used by CITE is designed to minimize such problems, they are not eliminated. Test cases may fail for a variety of reasons other than product defects. These reasons can be grouped into the following basic classes:

- Incorrect expected results.
- Test coding errors
- Incorrect assumptions by the test engineer.
- System anomalies

Because of this, it is important that each test failure be reviewed and analyzed to determine if a product defect was detected or if the test failed for other reasons. CITE provides a graphical interface for reviewing test failures which permits rapid test update to correct test defects, as well as permitting a test to be re-run under the same conditions in which it failed to ensure that the test passes when the cause of the failure is corrected.

2.3.1 KNOWN FAILURES

Frequently, a test fails due to a known problem in the product under test. To avoid review of failures which are known problems, the CITE failure review tools, *tdpp* and *tdreview*, can consult a database of known failures with which to filter the execution report.

Since the execution of tests known to fail may sometimes use valuable system resources that would be better spent executing tests that are expected to pass, facilities exist to prevent the execution of tests which are known to fail.

2.4 INTERACTIVE SIMULATION

Some products or utilities require interactive use for proper testing. For example, interactive programs such as *xterm* and *passwd*, cannot be tested by command line invocation alone. To accommodate such programs, CITE provides utilities which allow an interaction with a program to be described in a script which can then be executed automatically as though a human were performing the interaction.

2.4.1 X WINDOWS INTERACTION

The testing of X windows applications is especially problematic because an application interface can be easily changed. For example, a menu item may become a button on an application window. To solve this problem, CITE provides several X windows test utilities. [Min92] describes these utilities in detail.

2.5 TEST GENERATION

For many classes of test development [Ost88] and [Mau90], automated test generation can significantly improve the quality of a test suite as well as the time to development for a test suite. CITE includes two utilities to automatically generate tests. The first, *tsl*, was developed based on the ideas presented in [Ost88]. The formal specification language was extended to include actions which are executed as each equivalence class in each functional category is chosen for a test case. In this way, the specification compiler not only produces a list of tests that should be written according to the specification, it executes the appropriate actions and generates the tests. The second, *dgl*, is from the public domain utility presented in [Mau90] and uses augmented context-free grammars to generate tests.

2.6 TEST SOURCE CONTROL

The final element of CITE is the test source control system. Test suites are frequently composed of many hundreds of files. To efficiently manage changes to suites while reducing the disk space usage of a suite, CITE provides utilities to maintain separate revisions of test suites as tarred and compressed archives. Any revision can be retrieved, and revisions can be tagged to indicate the version of a test suite that was used to qualify a specific version of a product. In this way, the version of a test suite used to qualify a specific product revision can be easily retrieved whenever necessary.

3. RULE BASED TESTING

Each CITE test is associated with a *rule*. A *rule* is a script-like program which defines the sequence of events which compose a test and the results to verify for each event. The variables used in the *rule* define the set of information that must be provided by each test, and a set of information which may be specified each time tests are executed. The test supplies the specific information for the events defined in the rule.

For example, an extremely simple rule might define a sequence of two events: compile a file with some set of flags, verifying the stderr and return code of the compiler; and execute the executable that results from that compilation, verifying the stderr, stdout, and return code of the executable. Each test case using this rule would then specify the file name to compile and the set of flags to be passed to the compiler.

All of the information regarding a test case, including the name of the rule used by the test case is stored in the basesuite database.

3.1 ADVANTAGES OF RULES

The rules-based approach offers many advantages over more traditional test execution systems such as OSF's Test Environment Toolkit (TET)[OSF92], which require that each test case carry with it a script or a program to perform the necessary executions and results verification.

Because tests share rules, a rule can be written and debugged, thereby eliminating spurious test failures caused by an error in the script or program used to execute the test and verify the result. Extensive testing of the CITE utilities reduces the instances of test failures due to an error in the utilities. In this way, the sources of

test failures can be reduced to the product under test or the actual test code.

Using rules, storage requirements for a test suite are minimized because each test case need only consist of the files which are unique to it and the associated database entries

The maintenance of tests is also simplified. A rule can be modified to provide every test which uses it with new capabilities; test information can be modified easily by editing a template obtained from a database query. Expected results can be automatically updated for large sets of tests at a time because results are stored separately from the event sequence.

Tests are reusable, both from release to release of a product and by other products. For example, the compiler test suites at Convex are used to test not only the procedural compilers, but also the debugger, the performance analyzer, and the interprocedural compiler. In each instance, a simple change to a single rule allowed 90% of the existing test suite to be used without further modification.

Because virtually anything can be done within a rule, including the invocation of other system commands, the rule-based test definition allows CITE to be truly general-purpose.

3.2 THE RULE LANGUAGE

Rules are defined using a simple programming language. The syntax is similar to the UNIX C-shell syntax: variables are declared first, followed by the statements that make up the events which are to take place when a test which uses the rule is executed. A full description of the syntax of the rule language is beyond the scope of this paper, though the example in Figure 2 should accurately convey the basic use of the language. To enhance the expressiveness of the language, rules are pre-processed using the system C preprocessor in ANSI mode.

3.2.1 VARIABLES

Variables in rules must be declared, with each declaration specifying a type and a binding class. Although all variables contain string data, there are three possible types:

- file—values for variables with this type are file names needed by the test.
- environ—variables with this type can obtain their initial value from the invoking user's environment and their value affects the environment of subprocesses created by a test.
- string—variables with this type contain a string to which no special significance is attached.

Each variable has one of three binding classes, which specify at what point the variable is assigned its initial value:

- global—the initial value of the variable is defined by each test case. This is the mechanism by which a test case supplies specific information to the rule's general event sequence. The values of global variables are stored in the test suite database.
- export—the initial value of the variable is set when a test case is executed, either a default value specified by the rule, or a

FIGURE 2: EXAMPLE RULE

```
rule comptest
%the following variables can be changed at run time
 string export cc = "/bin/cc" /cc compiler to use/;
 string export Bflag = "" /-B path to use/;
 string export cflags = "" /flags to pass to compiler/;
 string export optimizations = "-no -O0 -O1 -O2 -O3" /optimization levels to test/;

%the following variables are set by the testcase
 file global src_code /Source files to compile/;
 string global flags /flags to pass to the compiler/;
 file global include_files /other files needed in the execution dir/;
 string global arguments /arguments for executable/;

%the following variables are set during execution
 string local executable /executable name/;
 string local opt /optimization under test/;

%compile and execute the resulting executable for each optimization level
 foreach opt ($optimizations)
    %create an executable name
    strcat($executable,$TESTNAME,".e",$opt);

    %compile the source(s), verify the compiler
    prog compile {rc,stderr:noncritical}
       $cc $Bflag $opt "-o" $executable $flags $cflags $src_code;

    %run the executable and check outputs
    prog execute {rc, stdout, stderr}
       $executable $arguments;
 endfor;
endrule;
```

value specified by the user when the test driver was invoked.

- local—the value is defined during the execution of the test. Variables of this binding class are temporary variables to simplify the definition of a rule.

Variable declarations are required to include a "comment" describing the use of the variable in the rule. This is not just enforcement of good programming practice, the comments are used to produce templates for test definition and to allow the test tools to supply context-sensitive help information regarding the options available when executing specific test cases.

3.2.2 STATEMENTS

The rule language supports most standard programming constructs, such as iterating over a list of values (*foreach*), conditional branching (*if-else*), and multi-way branch (*switch*). In addition, statements to manipulate strings are provided (*strcat*, *strsub*) as well as statements to add information to the report file (*printf*) and to call system commands (*system*).

While a more general purpose language would implement more extensive looping constructs, we have found the *foreach* statement sufficient for all testing currently done at Convex.

The key to the rule language is the *prog* statement, which defines an event, specifies the results to check, and the execution conditions for the event. The *prog* statement defines conditions such as the stacksize limit, the CPU time limit, the concurrency limit and scheduling of a process. A *prog* can be defined to verify any combination of the following possible results of a process:

- return code (exit status)
- standard output
- standard error
- cpu time spent executing user code.

Results can be defined as *critical*, which causes termination of the testcase if the verification fails, or *noncritical*, which allows the test case to continue execution after a failure so that system state can be restored by statements in the rule following the failing *prog* statement.

3.3 RULES AND THE TEST DATABASE

Each basesuite contains a database, which is composed of two directories for storing expected output (stdout and stderr), a single directory for storing files used by tests, and a dbm database for storing information about tests, such as the values of global variables.

As a test is executed, the statements in the rule used by the test are executed using the information about the test contained in the database.

4. AN EXAMPLE: TESTING A COMPILER

An example will help to clarify the use of CITE for automated testing. The rule presented in this example is a simplification of the rule used by the compiler test group at Convex. It is intended only to demonstrate the use of CITE, not as an example of a production rule.

4.1 COMPILER SUMMARY

To fully understand the following example, it will help to understand the Convex compiler technology. The procedural compilers consist of a driver program which invokes the actual parser and back end and calls the link editor with the appropriate system libraries. The compiler driver accepts a -B argument to indicate the location of the other executables used by the driver. In addition, the compilers have five possible optimization levels, from no optimization (-no) through vectorization and parallelization (-O3).

4.2 A RULE FOR COMPILER TESTS

In order to take best advantage of the CITE technology, it is best to have tests use as few different rules as possible. This implies that rules should be as general as practical while remaining understandable and readable. The rule shown in Figure 2 is probably adequate for 90% of C compiler tests. Keywords are boldfaced.

The variables declared **export** allow tests using this rule to use an alternate compiler (to test a compiler under development), to select the optimizations to run at (if a change was made to code executed only when parallelizing, there is no point to running at optimization levels below -O3), and to specify special flags to the compiler (to test a new compiler option). The variables declared **global** are used to allow the test to specify what files should be compiled, what files should be copied to the execution directory, what options should always be passed to the compiler, and what options should be passed to the executable generated by the compiler.

The statements loop across all desired optimizations, creating an executable name based on the test name and the current optimization level. The compiler is then invoked and the return code and error output of the compiler are verified against the expected results. If the output comparison fails, the test will fail, but the remaining statements will continue to be executed. If, however, the return code comparison fails, the test will fail and no further statements will be executed. Finally, the executable produced by the compiler is executed with the arguments specified by the test and the output and return code of the test are verified. If any results comparison fails, the test will fail and no further iterations of the loop will be executed.

4.3 CREATING A TEST

The first program compiled by most compilers is generally the classic "hello world" program advocated by [Ker78]. So the first test we create will be that program, in a file called **test1.c**. We will assume that a basesuite with the rule shown in Figure 2 is available and is the current working directory. The *tded* utility obtains information from the test suite about a test case or a rule and invokes the editor with the template obtained. Figure 3 shows the edited template, with the original template shown in bold. Note that the "comment" text of the global variables is used as a prompt in the template. When the editor is exited, the edited file is used to modify the database, in this example, the test **test1** is created, the file it requires (test1.c) is copied into the directory used as a repository for the files used by tests, information about each variable, the test description, and the user and group information are stored in the test database. The alert reader will note that

FIGURE 3: TEST EDIT TEMPLATE

```
Testcase name  test1
Rule  comptest
User:test
Group swtst
Testcase Description
test the classic hello world program

Lists
:Source files to compile· test1.c
·flags to pass to the compiler  -nw
other files needed in the execution dir
·arguments for executable.
command TESTNAME p compile -no '/bin/cc -no -o TESTNAME.e-no'
  rc  0
  stderr  /dev/null
·command TESTNAME.p.execute -no 'TESTNAME.e-no'
  rc  0
  stdout. /dev/null
  stderr  /dev/null
command TESTNAME.p.compile -O0 'bin/cc -no -o TESTNAME e-O0'
  rc  0
 .stderr· /dev/null
command TESTNAME.p.execute.-O0 'TESTNAME.e-O0'
 :rc  0
 .stdout  /dev/null
 .stderr  /dev/null
.command TESTNAME.p.compile.-O1 '/bin/cc -no -o TESTNAME.e-O1'
 ·rc: 0
  stderr: /dev/null
 command TESTNAME.p.execute -O1 'TESTNAME e-O1'
 ·rc  0
  stdout: /dev/null
 ·stderr  /dev/null
:command TESTNAME.p.compile -O2 '/bin/cc -no -o TESTNAME.e-O2'
 .rc: 0
 :stderr· /dev/null
 command TESTNAME.p.execute.-O2 'TESTNAME.e-O2'
  rc· 0
  stdout: /dev/null
  stderr: /dev/null
·command TESTNAME.p.compile -O3 '/bin/cc -no -o TESTNAME.e-O3'
 .rc  0
  stderr  /dev/null
·command TESTNAME.p.execute -O3 'TESTNAME.e-O3'
  rc. 0
  stdout  /dev/null
 .stderr  /dev/null
```

expected results information was not changed. For the return code, this is fine, a successful compiler execution will exit with a 0, normally the executable should also exit with a zero return code For the output of the compiler, and the executable itself, the empty file (/dev/null) is probably not the expected output; however, it is usually easiest to automatically generate the results rather than to pre-generate results and specify the results file when creating the test.

4.4 GENERATING EXPECTED RESULTS

The next step is to generate expected results. The *td* utility is used to execute the test and save the actual results as the expected results. The following command line will accomplish this:

```
td -e /tmp -z all   test1
```

If the test were again edited with *tded*, the expected results information would be filled in, as shown in Figure 4.

Once generated of course, the results must be carefully reviewed by the test development engineer to ensure their correctness. While this is an error-prone portion of the process, the risk is minimized by following certain conventions, such as

- Test executables verify results (such as numerics) against results computed differently internally, rather than simply printing them.
- Output comparison is done using "masks" which ignore unimportant output.
- Output is minimized, for example, the compiler has a special output mode that simply provides a count of the optimizations performed, rather than a full optimization report. This mode is used for all tests except those that are designed to verify the customer visible optimization report.

FIGURE 4: PARTIAL EXPECTED RESULTS INFORMATION AFTER GENERATIING RESULTS

```
:command test1.p.compile.-no '/bin/cc -no -o test1.e-no -nw test1.c'
 :rc. 0
 :stderr: /dev/null
:command test1.p.execute.-no 'test1.e-no'
  rc. 0
 :stdout: Stdout/test1.p.execute.-no.so
 ·stderr: /dev/null
·command test1.p.compile.-O0 '/bin/cc -O0 -o test1.e-O0 -nw test1.c'
 :rc: 0
 :stderr. /dev/null
:command test1.p.execute.-O0 'test1.e-O0'
 :rc: 0
 :stdout. Stdout/test1.p.execute.-O0.so
 :stderr: /dev/null
:command test1.p.compile.-O1 '/bin/cc -O1 -o test1.e-O1 -nw test1.c'
 :rc: 0
 :stderr: /dev/null
:command test1.p.execute.-O1 'test1.e-O1'
 .rc: 0
 :stdout· Stdout/test1.p.execute.-O1.so
 :stderr· /dev/null
:command test1.p.compile.-O2 '/bin/cc -O2 -o test1.e-O2 -nw test1.c'
 :rc: 0
 :stderr: /dev/null
.command test1.p.execute.-O2 'test1.e-O2'
 :rc: 0
```

FIGURE 5: SAMPLE REPORT FROM TD

```
Execution started: Mon Jan 11 18.27:22 1993
Hostname: foo
Starting directory: /mnt/pvogel/doc/ISSTA/example
Execution directory· /tmp
Starting suite. /mnt/pvogel/doc/ISSTA/example
Initial UID: pvogel
Initial GID: swtst
/vmunix version: 10.0.3.0
TD executable: /usr/local/bin/newtd/td
TD version: 4.2.0.1
Command line arguments. '-e' '/tmp' '-x' 'cc' '/usr/convex/fc' '.' 'test1'
Machine type: CONVEX C2xx multiprocessor
Number of heads: 2
IEEE Hardware: Available
Default FP Mode: NATIVE
TD PID· 15068


Started /mnt/pvogel/doc/ISSTA/example Mon Jan 11 18:27·23 1993
example/test1:FAILED;critical; prog=test1.p compile.-no; rc act=1 exp=0;
 command '/usr/convex/fc -no -o test1.e-no -nw test1.c'
 return code:
    expected 0
    actual 1
Finished /mnt/pvogel/doc/ISSTA/example Mon Jan 11 18:27:25 1993

Execution stopped: Mon Jan 11 18·27:25 1993
```

4.5 EXECUTING THE TEST

Once expected results have been generated and verified, the test can be run to verify that a compiler works. Since the test was run to generate expected results, we can assume that the compiler works, however, an alternative compiler may not. For example, the following command line will execute the test using the FORTRAN compiler instead of the C compiler

```
td -e /tmp -x cc fc . test1
```

The result is of course, a compiler error, which is reported as a failure (though in this case, the failure was with the engineer that attempted to run a C test against a FORTRAN compiler!). The report from *td* is shown in Figure 5.

4.6 ANALYZING THE RESULTS

Given the report in Figure 5, indicating that a test failed, it is necessary to analyze the failure. The report indicates that the test

failed because the compiler exited with an unexpected return code. The report indicates the *prog* iteration that failed (test1.p.compile.-no). In the execution directory (/tmp) the error output of the compiler is stored in the file with the prog iteration followed by **.SE**. A quick look at this file shows us this error

```
Cannot compile c-language file 'test1.c'
```

A quick look at the report file shows that the command executed was an *fc* command instead of a *cc* command, and the reason was the definition of the **cc** export variable, and the failure can be chalked up to an error on the part of the engineer running the test.

5. EXPERIENCES WITH CITE

5.1 ORIGINS

Convex was founded in 1982 by a small group of people with the vision of creating a "mini-supercomputer." It was recognized that with the limited resources available, hand testing would not be a viable option. A shell script, *swtest*, was written as a prototype for the test driver now known as *td*, in order to better define ad refine the requirements.

5.2 EVOLUTION

Based on the experiences with *swtest*, the first elements of CITE, the test driver and associated tools for adding tests to suites were developed, along with tools for performing test coverage analysis. This version of the tools underwent steady enhancement as Convex grew and software projects became more ambitious with a commensurate need to increase the level of testing. Additional hardware platforms also drove many of the enhancements to the tools, such as the ability to partition expected results for tests depending on the characteristics of the platform on which they were executed.

Although the tools met the basic needs of automated test execution, there was one major problem.

- The suite database was easily corrupted, changing a rule used by a test in the database virtually guaranteed an unusable database.

This problem resulted in rules scattered throughout a test suite hierarchy which had the same name but worked slightly differently from suite to suite—severely impeding the maintainability of the suite.

5.3 THE PRESENT GENERATION

In 1991, recognizing the need to resolve the problem with the database in order to improve the maintainability of suites containing over 1 million lines of code and several hundred distinct rules, an effort was launched to overhaul the database mechanism to allow rules to be modified without adversely affecting the databases using those rules.

The result was the CITE described in this paper, in which one large rule (1200 lines) is used to manage virtually every test for all compiler products.

5.4 RESULTS

Because the test environment relieves test developers from the burden of writing scripts to drive the execution of tests, Convex is able to achieve high levels of test coverage (75% or higher) despite a developer to test engineer ratio that is the inverse of industry norms (2-5 developers per test engineer). Further, in the groups with small development to test engineer ratios, the development group is importing large bodies of code from third parties such as OSF and Hewlett Packard.

Portions of each test suite are executed against the current development version of the product they are intended to test every night, with the entire test suite getting executed over the course of one to two weeks. As a result, defects introduced during product development have a high likelihood of being detected within a week of their introduction. In this way, Convex has dramatically improved the quality of its software products while reducing the development cost of those products. In most software development areas, defects reported by customers become permanent regression tests in the product test suite. In addition, the regular execution of the test suites against the current product helps to detect bugs in the test suites in a timely fashion, providing a high level of confidence in both the product and the test suites at qualification time.

When development of the interprocedural optimizer commenced, the interface to execute the interprocedural compiler was defined to be different than the procedural compiler interface. Test engineers were able to use the entire FORTRAN and C test suites to verify the new compiler simply by editing the rule used by the test suites, saving hundreds of staff-hours by not having to edit each individual test case.

6. CONCLUSION

The ability to automate test execution has long been recognized as a significant contributor to the software development process. We believe CITE re-defines the state of the art, the rules-based test execution simplifies maintenance of tests and helps to reduce spurious test failures.

Some areas of CITE still require further work, and development of new capabilities is ongoing at Convex. Many test systems are able to retain statistics regarding the history of test cases, while CITE does not yet support that functionality. The tools for testing X windows applications would benefit greatly from the ability to compare bitmaps with certain tolerance levels as well as the ability to recognize simple transformations of expected bitmaps, such as scaling. As parallel processing technologies are explored, the management of resources used by tests to avoid resource conflicts will become increasingly important.

REFERENCES

[Ker78]
Kernighan, B.W., Ritchie, D.M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1978.

[Lib92]
Libes, Don, Expect source distribution, available by ftp, 1992

[Mar93]
Marick, Brian, GCT source distribution, available by ftp,

1993.

[Mat91]

Mathur, A. P., "On the Effectiveness of Data Flow Testing," *Proceedings Quality Week 1991*, San Francisco, 1991.

[Mau90]

Maurer, P.M., "Generating Test Data with Enhanced Context-Free Grammars," *IEEE Software*, pp 50-55, July 1990.

[OSF92]

Open Software Foundation, UNIX International, X/ Open Company, *Test Environment Toolkit: Architectural, Functional, and Interface Specification*, Unpublished, available by ftp, 1992.

[Ost88]

Ostrand, T. J., Balcer, M. J., "The Category-Partition Method for Specifying and Generating Functional Tests" *Communications of the ACM*, pp. 676-686, June 1988.

[Sit91]

Sittenauer, S., Daich, G., Samson, D., Dyer, D., Price, G., Hugie, J., Peterson, G., *Software Test Tool Report*, Software Technology Support Center, Hill AFB, Utah, 1991.