ENVIRONMENTS FOR TESTING SOFTWARE SYSTEMS

Kent C. Archie and Robert E. McLear, III

***Kent C. Archie** is a member of technical staff in the Data Networking Decisions and Design Systems department in the AT&T Information Systems in Naperville, Illinois. He is currently involved in developing test management systems, including the Buster system described in this paper. Mr. Archie joined AT&T in 1985 with a B.S. in mathematics and an M.S in computer science from the University of Wisconsin at Madison. **Robert E. McLear, III** is a supervisor in the Switching Processor and Operating System Development department of AT&T Bell Laboratories, Naperville, Illinois. He and his group are responsible for the operating system and database components of the RTR operating system. He has a B.S. in mathematics and computer science and an M.S. in computer science,*

(continued on page 75)

As software becomes a larger part of many products, thorough testing becomes increasingly important. The testing process begins before the requirements are finished and continues until the product is discontinued. In this paper, we discuss the software testing process, its difficulties and costs. We set forth the features of a testing support system and how the Buster test support system—currently in use at over 50 sites throughout AT&T—helps realize those features. We also discuss further steps to increase the effectiveness of software testing and increase the value of the final product.

Introduction

In his 1983 book, *Software Testing Techniques*, Boris Beizer claims that "Testing consumes at least half of the labor required to produce a working program."[1] In mid-1985, we estimated that 30 percent of all project staff for the upcoming AT&T 3B4000 computer release needed to be involved in system testing and system quality assurance. But if we were to include the efforts of *unit testing*, the 50 percent Beizer suggested would be a low estimate. We anticipated that following the initial release there would be several releases (or generics) of the product with a high degree of code reuse. If a test suite had to be developed from scratch for each release, the cost of testing would spiral to unacceptably high levels. We needed a process improvement to speed up testing, reduce staffing requirements, and maintain the effectiveness of the testing effort.

A "test working group" was chartered to explore this problem and propose a timely solution for the start of the AT&T 3B4000 computer system test. The deadline was six months away, and the 3B4000 project was committing to use the process and test environment created by the working group.

Many of the problems we encountered in 1985 still face testers today. We will discuss those problems, and then suggest how a standard test environment can help solve them.

What Is And Is Not Testing

There are as many definitions for *testing* as there are testers. However, there are common axioms familiar to most testers:

- Development always leaves bugs in the product.
- When testing "completes," it is desirable that all potential customer-observable bugs have been found and repaired.
- Customer-observed bugs cost the company money—in terms of both warranty cost and market share.
- Although testing adds cost to a product, it also adds value because it prevents future warranty and market share expenses and loss.

We conclude that testing must detect as many customer-observable bugs as possible, and at the lowest cost. It must also verify that features and functionalities that worked properly in the past will continue to do so in the future.

Software for testing the product resembles the deliverable software in that it is expensive to create and maintain. Often, test cases are developed on an ad hoc basis for a project, but then discarded when that project is finished. As with deliverable code, the benefits of reuse and the accumulation of test cases both are lost. Note that reusing an existing regression test suite, particularly when the coverage of product code or functionality is low, can lead to a false sense of quality in the delivered product.

Testing, far from being something tacked on at the end of the software development process, is integral to that process. It is an ongoing activity, begun before the requirements are finished and completed only when the product is discontinued.

The Problem

The problems we found in 1985 centered around the lack of a *standard process* for testing. That is, each tester had his or her own way of doing things. This factor alone created a major expense: the amount of skilled labor required to create, administer, execute, and interpret tests. With no standard testing process, testers were trained by using the "apprentice" system. Information about how to write test cases, execute tests, and report results was passed on by oral tradition. The lack of a standard testing process manifested itself in other ways:

- There were no uniform standards for test cases on the 3B4000 project, only some time-tested guidelines for writing high-level test designs.
- Some test developers wrote in the C language, others wrote in `shell`. Because the test cases were written in different styles and programming languages, it was hard to move testers between teams.
- Some testers did their testing black box fashion—i.e., basing their testing only on the product's requirements and documentation—and others wrote test cases white box fashion, using their knowledge of the product's design and implementation to guide their testing efforts.
- When reports were given by different testing teams, each team counted tests differently. It was difficult to determine how many tests had been run, how many had passed or failed, or how many remained to be run.
- Tests were not regularly reused or shared between releases and teams. They were stored in different formats and locations, and often were lost when the original testers left the test team.

We set out to standardize the testing process and implement that standard in time for the 3B4000 computer release. We envisioned a system that provided both test administration and automation in support of the standardized process. In addition to addressing the specific problems already described, the standard process would help alleviate the effect of excessive staff turnover sometimes present in large projects with extended schedules.

For the administration system, we saw the primary problem as one of test case *resource management.* We needed to know the following about the test cases:

- What test cases were available
- Where did they reside

- What was the status of the test cases.

The team first developed a list of problems facing testers, then drew up a list of possible solutions to those problems.[3] The two lists were compared, and the solutions that solved the most problems became the requirements for the new testing environment. We conducted an investigation of existing systems to find one we could use. We found that existing test management systems were inadequate for some of the following reasons:

- Some required a particular class of terminal or a specific computer, and therefore were not portable between and among systems.
- Some provided test execution but did not provide tracking or reporting. Others provided tracking and reports but did not support test automation.
- Some systems consisted of one large program that made them impossible to customize. Others had a difficult user interface.

When we determined that none met enough of our criteria, we set out to develop one that would.

The Ideal Testing Environment

The number of features in an ideal testing environment probably is equal to the number of testers asked. Below are the core requirements we developed for a testing system.

Standardized Format. Tester training time is greatly reduced if all testing groups use the same test format and record results the same way. Sharing test cases within, between, and among testing groups is also facilitated.

The standard test format should include documentation to describe the purpose of the test and how to execute it. The tests should be run using a standard execution mechanism. The combination of documentation and standardized execution will provide a stable test environment that will make learning how to run tests easier.

Early in this effort, the test working group agreed to use the IEEE 829 Test Documentation Standard[2] as the basis for the software testing process documentation, and the test case description in particular.

Test Searching. Determining if there are tests that can be used or modified to suit a new project usually involves paging through listings or searching files. A database that summarizes the information in the test files would greatly improve the effort to identify tests that meet the specific search criteria. Testers can search the databases of other projects to locate tests that could be reused on their current project.

Ease of Use. Though system ease of use is a subjective measurement, it can have a quantifiable—i.e., objective—effect on a tester's productivity. Testers will not use a system that is too hard or too clumsy if they believe it will slow down their work. Thus, the cost of developing the testing system will be lost. The tool must be flexible enough to handle circumstances unforeseen by the developers, but remain simple enough for first-time or occasional users. A collection of interacting tools will provide this flexibility.

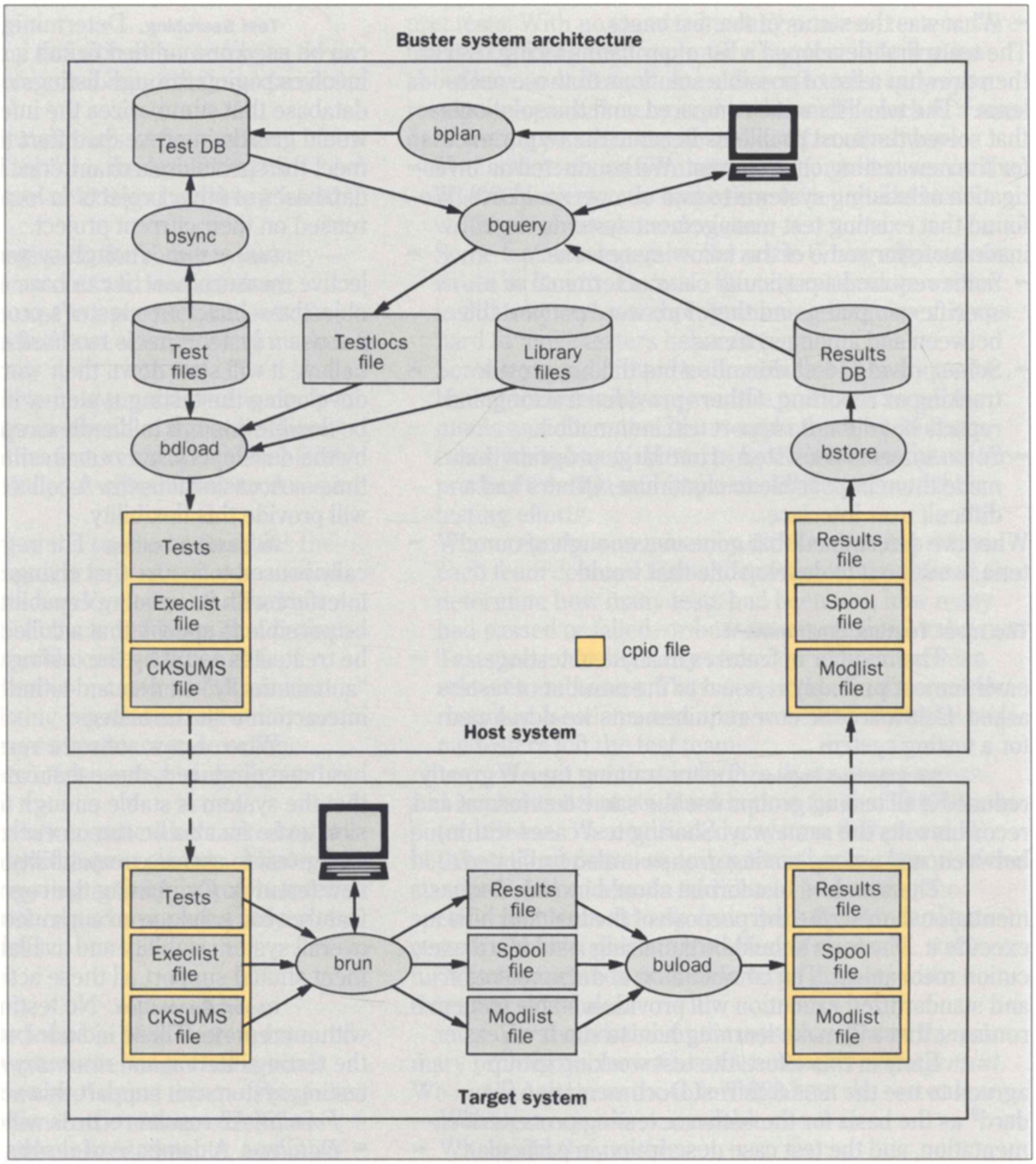
Regression Testing. The regression test set typically is used to insure that changes to a feature do not interfere with its existing capabilities. It should therefore be possible to specify that a collection of test cases must be treated as a unit by the testing system, and executed "automatically" on demand—that is, with little or no interaction with the tester.

When a new software version for some feature has been produced, these test cases can be run to verify that the system is stable enough for testing. The regression tests can also be run on each release of the system under test to assess compatibility between the old and new features. Combining the regression sets for several features can produce an automated test package to test overall system stability and availability. The test environment should support all these activities.

Report Generation. No testing effort is complete without reports. These include both *status reports* during the testing interval and *summary reports* at the end. The testing system can support this activity in several ways.

- *Format.* All results records will have identical formats.
- *Database.* A database of results records makes it

Figure 1. Buster system architecture



easier to generate and interpret complex reports involving information about the test cases and results.

- *Analysis.* If the records contain execution time data from the test runs, they can be used to analyze feature reliability.

Reports are only as good as the accuracy of the underlying data. Thus, the test system should gather the needed information as automatically and transparently as possible to guarantee accurate data.

Reliability of the Testing System. It is essential that the testing system be reliable. A tester should be sure that any failure was caused by the system under test, not by the testing system. Thus, the test system must be as simple as possible. In the context of automated test execution and data collection, the test system must not interfere with the system under test. A simple design would also allow it to be used in several situations by adding software or hardware attachments as they are needed. We thought it a better strategy to allow testers to add components than to develop potentially costly "workarounds" to circumvent it.

Support and Training. The system will reduce tester training because the knowledge the tester gathers is portable across features and across testing organizations. A simple test system also allows for faster tester startup when testers are newly assigned to a feature.

Test Code Source Management. Control of the software development process is greatly facilitated by source code management systems. A similar source code control technique can be applied to test case source code management, and will allow incremental modifications of test cases.

Planning and Predicting. The testing system can support test planning in several ways. It can track test cases—in a way similar to the feature code—as test cases move through the planning, developing, testing, and run phases. The tester should be able to start entering planning information as soon as requirements are available.

The test system should support tracking individual test cases as well as test packages. The tester should be able to define packages, add test cases to them, run a package as a unit, and collect information on the collective results of the test cases in a package.

Test Ordering. In many cases, the tester must run a test set in a specific order. One example of this is protocol testing, where the test order would be determined by a finite state machine. The test system should allow a tester to specify the order of a test set. If one test is requested, the rest must come with it and be run in the order specified.

The Buster Test Management System

The analysis of the test working group mentioned earlier led to developing the Buster test management system in 1985. The development team consisted of both testers and tools providers. A system prototype was developed and tested by the 3B4000 computer project. As this prototype was used, we learned more about testers' needs, and were able to modify the prototype until it became the production Buster system.

Several factors led to why Buster is successful. First, in the sense that the development team was also a user of its own system, it can be said that Buster was used to test itself. The development team had to respond quickly to problems reported by the user community because it was part of that community. Second, we had members of the user community as part of the design and development team. Thus, the team always knew what its customers wanted because there was considerable overlap of roles between development and system users. Third, the heavy use of prototypes and a short loop between the development team to the testing community prevented Buster from diverging from the needs of the testers.

Buster Design and Philosophy. Monolithic testing systems are too inflexible to meet the needs of diverse testing organizations. Buster was designed as a modular collection of small interacting tools. The most important design principal was to ***keep it simple***. Figure 1 shows the overall architecture of the system.

Panel 1. Example Test Template

ID:	A unique test identifier
Origin:	Network address of the test's origin
Type:	Automatic, manual, or TTY
Object:	Feature being tested
Contact:	Author or maintainer of the test
Keywords:	Subject areas for this test
Purpose:	Part of the feature being tested
Reqt:	Requirement number being tested
Method:	Description of how the test will be done
Doc:	Document describing the feature or test
Package:	List of packages associated with the test
Library:	Name of test code libraries needed by this test
Shell:	Shell to use to run test code sections
Sconfig:	What software must be present for the test to work
Hconfig:	What hardware must be present for the test to work
Comment:	Any other documentation
Count:	Number of testcases within this test
Stime:	Time needed to run the setup section
Ptime:	Time needed to run the procedure section
Ctime:	Time needed to run the cleanup section
Setup:	Code to be executed to set up the test environment
Procedure:	Code that performs the test
Cleanup:	Code to restore the environment to its original condition.

Information is passed between Buster modules through simple files and databases; these interfaces are well defined. Buster's modularity brings it in line with the UNIX® system philosophy of combining tools to complete a task. Users can replace Buster components with their own programs if doing so meets their needs.

From the outset, achieving reliability was a major goal in Buster's design. The multi-component design is valuable, because each tool is small and has limited functions. This made it possible to isolate each component and test it independently. Often, Buster needs to be run in an operating environment that is still under development. Thus, keeping the execution component simple allowed Buster minimal interaction with the operating system being tested.

This remainder of this section will describe how the Buster components are used during a test's life cycle.

Testscript Creation. A Buster *testscript* consists of several text sections identified by reserved keywords. (Panel 1 shows the sections and their definitions. Panel 2 is a example of a completed testscript.) The format is based on the IEEE 829 Test Documentation standard and allows for inclusion of test case documentation as well as the test code. Buster does not impose a rigid structure on the contents of the documentation sections. This allows different testing organizations to use different documentation techniques while still maintaining a common format.

When the requirements for a deliverable feature have been stabilized, test development can begin. The tester uses `bplan` to create the initial entry into the test database. `Bplan` prompts the user for some of the sections listed in Figure 1 and creates a database record. It also creates a testscript template with some sections filled in. The tester can then complete the testscript, including the SETUP, PROCEDURE and CLEANUP sections (as noted in the IEEE standard) that contain the UNIX system `shell` code to perform the test. The testscript is then moved into a directory in the test storage hierarchy (TSH) where the testing organization is free to construct the hierarchy as it wishes. All Buster test cases are stored as directories in the TSH. One file in the directory is the testscript; the others can be data files, executable code, documentation, or anything else needed for the test.

When the testscript is complete, or whenever it is changed, `bsync` is used to update the information in the test database. The testscript is considered the repository of all information about the test.

Panel 2. Completed Buster Testscript

```
ID: demo.twg.01
ORIGIN: 3b.demo.twg.01
TYPE: a
OBJECT: basename(1)
DOC:
KEYWORDS: command
PURPOSE: To test the basename(1) command
REQT:
METHOD: Create a data file containing pathnames for certain
        files.  Read the data file, one line at a time and
        pass the pathname to basename(1).  Compare the
        output of basename to what was expected and exit
        on any failures. Continue until the file is exhausted.
LIBRARY:
SCONFIG:
HCONFIG:
COMMENT:
COUNT: 1
STIME: 5
PTIME: 5
CTIME: 30
SETUP:
cat >DataFile <<!EndOfDataFile
/d1/d2/f1
/d1/d2/f2
/a/b/c
/a/b/d
!EndOfDataFile
PROCEDURE:
while true
do
        read PATHNAME || break
        F=`eval basename PATHNAME
        FILENAME=`echo {PATHNAME} | cut -f4 -d"/"`
        if [ $F != {FILENAME} ]
                then
        echo "Expected: {FILENAME},  Got: $F"
                fail
                exit 1
        fi
done <DataFile
pass
CLEANUP:
rm -f DataFile
```

Panel 3. Buster Test Output

```
#######################################################################
(1.2) BUSTER SESSION STARTING Tue Apr 12 10:16:25 1989
#######################################################################
STARTING STATUS:
---------------
SESSION NAME        : iwtom576864985
RESULT FILE         : RES.576864985
CONFIG              : unk
SPOOL FILE          : SPL.576864985
================== START TEST: demo.twg.01 ==================
----Id: demo.twg.01, Format: buster, Type: auto, Run: D-----
SETUP(timeout=5.00)-----------------------------------------
PROCEDURE(timeout=5.00)-------------------------------------
Passed
CLEANUP(timeout=30.00)-----------------------------------
-------- Results: 1 pass, 0 fail, 0 inc, 0 not run ---------
=================== END TEST: demo.twg.01 ===================
FORMATING RESULTS: RES.576864985
------------------------------------------------------------
ID          TYPE  SDATE    BRTIME PASS FAIL  INC  NR COMMENT
------------------------------------------------------------
demo.twg.01 D     04/12/88   0.08    1    0    0   0
------------------------------------------------------------
------------------------------------------------------------
TOTALS:                      0.08    1    0    0   0
------------------------------------------------------------
Total Test Scripts: 1                 Total Test Cases Run: 1
ENDING STATUS:
---------------
SESSION NAME        : iwtom576864985
RESULT FILE         : RES.576864985
SPOOL FILE          : SPL.576864985
CONFIG              : unk
#######################################################################
(1.2) BUSTER SESSION COMPLETED Tue Apr 12 10:16:34 1989
#######################################################################
```

Test Selection. The tester uses the `bquery` component to search the database and produce a list of the test IDs—the *testlocs* file—he or she wishes to run. The `bdload` component then packages the testscripts and any support files for the user to transport to the system under test.

There are many ways to move files between machines. Buster does not require any particular transport mechanism, allowing it to be used in any environment and over either floppy or tape networks. In order to maintain its flexibility in an ever-changing networking environment, Buster does not provide a transport mechanism.

`Bdload` also produces another list, *execlist*, to tell the Buster test system, `brun`, what test cases to run. It also generates a `checksums` file to determine if the test cases were altered on the system under test.

Running the Tests. In a larger system, test automation would provide for computer-assisted test case creation. Instead, Buster provides a simple test automation facility through the `brun` test executor. The `brun` command is all that is needed to run one or more tests and collect the results. `Brun` executes—in order—the SETUP, PROCEDURE and CLEANUP sections of the testscript using a user-specified program, by default the UNIX system `shell` program. The code in those sections is responsible for determining the result of the test sequence, and calls one of the Buster routines—`pass`, `fail`, or `inc` (for *inconclusive*)—to record the result of running the test. All input and output produced during the test run is also recorded. Panel 3 shows the results of running the test illustrated in Panel 2.

Once brun completes the test session, it produces a file with the database records for each test that was run, a file containing all input and output, and a list of the test cases modified while on the system under test. These files can be packaged by buload to be transported to the host computer.

Result Storage and Report Generation. The bstore command can take the results file produced by brun and insert the records in the results database. The results records contain:
- Test identifier
- Time and date of the run
- Result
- Machine name of the system under test
- How long the test took to run
- Any user comments.

Buster interacts with the UNIX system UNITY database management system to provide tools for report formatting. UNITY is used to contain the test information and the results records, and the testers are free to use UNITY's facilities to generate reports in any format they wish. UNITY's simplicity (i.e., all data is stored in American Standard Code for Information Exchange [ASCII] files) allows the tester to use an array of UNIX system tools such as awk and grep to analyze the results of testing.

Is Buster the Ideal Testing Environment?

Buster has been used successfully to test all AT&T 3B-series computers. It has been used to test the UNIX system itself, as well as software development tools and telephone switches.

Standardized format. The IEEE 829 standard has proved flexible enough to meet the needs of all Buster users. Test sharing between projects has not proved to be as common as we thought, but many projects report large time savings by reusing test cases from one release to the next. For example, test cases developed for testing UNIX system commands on the 3B2 computer were regularly reused on the 3B4000 computer project.

Test Searching. The UNITY database was chosen for its simplicity and because it was available as an in-house tool. However, the development team views UNITY as a temporary expedient while Buster is still in its earlier releases. The Buster developers are investigating the use of a different database with greater speed and ability to quickly process large test databases in future releases of the product.

Ease of Use. By using reasonable default values for the arguments of commands, and by making each component small, little training is needed to use Buster. Currently, a more consistent, simpler user interface for Buster is being developed. It will make use of a menu interface like commonly used in commercial software. The user's guide has been rewritten to present the information needed to run Buster in a more useful manner.

Buster's robust, multi-component design has allowed the different project teams to customize it around their needs. Indeed, nearly every project team has made some change to Buster either by replacing components or by building commands on top of it. One group, to use a different database, replaced every part of Buster except brun. This is a positive comment on Buster's robust design.

Because of Buster's use of a standard test case format, cases are insulated from the computing environment. Testers can assemble test cases from a variety of sources to create a test suite. At the same time, each project's testing method can differ according to the project's requirements. Project team developers and testers are never bound by a set of rules imposed by Buster's developers.

Regression Testing. Regression testing involves collecting a set of test cases to examine the stability of a feature. Buster supports this with the concept of *packages*. A Buster package is a named collection of test cases that are run together, and the package name can be used in test selection.

Reliability. As was mentioned before, the multicomponent design of Buster keeps the pieces small and simple. Keeping the execution component simple has allowed for minimal interaction with a highly complex operating system. Over the past four years, most of our error reports have come from the development group. We have experienced few error reports from the field.

Support and Training. Buster is supported by the central development team through a single help line. At the local level, each Buster site appoints an administrator who becomes the local Buster subject matter expert and first-line problem solver. Thus, simple problems can be handled quickly, and more complex problems can be referred to a single point of contact. Manuals and some sample report generation techniques are provided for both the tester and Buster administrator. We also provide videotapes for a half-day class in Buster use.

Test Code Source Management. At its present stage of development, Buster is not directly associated with an internal change control system. If a testing organization wishes to use change control, it needs to extract the test cases from Buster before using the `bdload` component. We are investigating future connectivity between Buster and the software product administration (SABLE) change management system.

Planning and Predicting. Buster allows a tester to create entries in the test database at any stage of the testing life cycle. A partial record can be created as soon as identifiers are assigned. As more information is acquired, the record can be filled in.

Test Ordering. Support for test ordering is difficult because the ability of each test to run properly normally depends on the tests that precede—and follow—it. If tests are not run in the proper order, the system may respond with illegitimate test failures.

To provide such support, the test case would have to contain a description of the environment and context expected for the test case to run properly, and select an ordering of test cases that would satisfy the expected environment for each and every test case in the set. One of our design principles was to keep the test execution system simple. We decided to not support this need in the early versions of Buster.

In future releases, the user will be allowed to specify an ordering within a package of test cases. When this package is selected, the test cases will be run in the specified order.

Successes Using Buster

Buster is being used by over 50 testing groups and is currently managing over 25,000 test cases. We discuss the experience of two of the larger users of Buster, and the effect it had on their work.

3B4000—Commercial UNIX. The 3B4000 computer project was Buster's first customer. Several members of the Buster design team also were members of the 3B4000 testing organization, and therefore were committed to using Buster from the outset of their testing effort. The quality of Buster's early releases was high enough so they could successfully test the product. By the time they were completed, they had written over 4,000 test cases. Using Buster, only one week was needed to run a regression test suite that formerly took three weeks.

The 3B4000 computer team developed a large set of tools built out of `shell` and UNITY commands to track the testing effort and generate reports. During their tests of the operating system, the team found several errors in the host operating system while they were installing `brun`. The Buster developers proceeded to develop a smaller version of `brun` that could run on systems whose operating system is still in a developmental phase.

System 75/85—Commercial Switching. The AT&T Definity® private branch exchange (PBX) system project received a copy of Buster for evaluation purposes. Within a short time, they had 14,000 test cases under Buster and were using a different database system that was more efficient when working with the volume of test cases they were generating. They were already using Relational

Technology's INGRES™ database management system for their work, and built applications to support the Buster architecture. The architecture was simple enough so modules of Buster could be replaced without assistance from the developers.

In addition to changing the database, the Definity PBX project team connected Buster to their existing custom test automation tool that used the PRAIRIE test execution language and its related Traffic Regression Test System (TARTS). This was a simple operation, although the two systems were quite different and the Buster source code was not available to the users. The tests were written to generate commands coded in the PRAIRIE language. These commands were used to control the test automation system connected to the switching system under test. The automated tests simulated telephone traffic.

The Definity PBX project had vastly different needs than previous Buster customers and was able to capitalize on the modular design of Buster to create a custom test management system, Test Support Information System (TSIS).

Conclusion

The environment we created saved the 3B4000 computer project 10 to 15 percent of the testing costs—3 to 5 percent of the overall project cost. The effort expended in building the Buster release used by the 3B4000 project was about the same as the effort saved. It paid for itself in one application and continues to provide savings to projects throughout the company. Other projects are realizing similar savings, and still more projects are beginning to invest either in using Buster, or in creating similar, customized testing support environments for their use.

There were several factors underlying this success. First, we had members of the user community on the design and development team. This kept us focused on the needs of the testers. Second, we developed and deployed a prototype of the system to use as a kind of executable requirements document. This insured that Buster was not only what customers asked for, it was also what they wanted. Third, we designed a modular architecture for Buster. This allowed it to be used in projects that have quite different testing needs. This architecture has withstood the test of five years of use.

We discovered Buster not only is a collection of tools, but also it is the implementation of a testing process—a process that consumes time and money. As we indicated, estimates run from the 30 percent we observed to the "at least half" estimate from Beizer and other sources. The potential payoff from continued investigation and improvement of the testing process and support environments is well worth the investment.

Acknowledgment

We wish to acknowledge and thank Y. Levendel of AT&T Bell Laboratories for his insight into the potential drawbacks of reusing an existing test suite, and for his careful attention to this article during the late stages of preparation.

References

1. Boris Beizer, *Software Testing Techniques*, Van Nostrand Reinhold Company, New York, 1983.
2. *Software Test Documentation*, *IEEE Software Engineering Standards*, ANSI/IEEE Standard 829-1983, American National Standards Institute/Institute of Electrical and Electronics Engineers, New York, 1983.
3. D. Wodarz, "An Automated Solution to Common Testing Problems," *Third National Communications Forum*, Chicago, Illinois, September 28, 1987.

Biographies (continued)

both from Purdue University, Lafayette, Indiana. He joined AT&T in 1979.

(Manuscript received December 18, 1989)