MOTHER: A Test Harness for a Project with Volatile Requirements

Joe Maybee

Graphic Printing and Imaging Division
Tektronix, Incorporated
Mail Stop 63-424
P.O. Box 1000 Wilsonville, Or. 97070
Usenet: maybee@pogo.WV.TEK.COM
Phone: 685-3572

ABSTRACT

This paper describes experiences with a test harness designed to quickly accommodate changing external requirements. This paper is intended to share experiences and ideas for automated testing with quality assurance engineers. The system under test is a real-time embedded system that controls the complex electromechanical mechanisms in a graphic printer.

Keywords and Phrases: real-time embedded systems, automated test methods, requirements testing, electromechanical systems.

Biographical: Joe Maybee is a Senior Software Engineer with the Graphic Printing and Imaging Division at Tektronix. Joe has been employed as an engineer with Tektronix since 1978, and has specialized in the design and implementation of real-time embedded systems during this time. Recently, Joe has focused on software quality in real-time embedded systems and the accurate definition of user requirements.

This paper was originally prepared for and presented at the *1991 Pacific Northwest Software Quality Conference*.

MOTHER: A Test Harness for a Project with Volatile Requirements

Joe Maybee

Graphic Printing and Imaging Division
Tektronix, Incorporated
Mail Stop 63-424
P.O. Box 1000 Wilsonville, Or. 97070
Usenet: maybee@pogo.WV.TEK.COM
Phone: 685-3572

1. Introduction

1.1 The basic nature of the problem to be solved: changing requirements

Software engineers have long based procedures and methods upon the existence of concise, immutable requirements. The truth of the matter is that requirements are seldom, if ever, the static, universal truths that we wish them to be.

In the classic software models, such as the *waterfall* model of software engineering, changing requirements cause a ripple effect that cause perturbations throughout the engineering process ‘‘pipeline’’.

Regardless of how we wish that requirements were unchanging, the brutal truth is that freezing a requirements specification to achieve a smooth, textbook software engineering process can result in an engineering process that produces a high quality product that no one wants to buy. The fact that requirements are volatile is a reality built into the environment in which today’s products must compete: customers’ requirements are changing at a faster rate than ever before.

This paper describes how one *Quality Assurance* (QA) group attacked the problem of changing requirements.

1.2 The nature of the problem from the QA viewpoint: changing tests

From the QA viewpoint, the problem becomes one of reacting to the changing requirements on demand. Providing facilities for quickly modifying, adding, or deleting the defined set of requirements *and* their associated tests allows the QA team to respond to changing requirements in a timely fashion.

Reacting quickly to change becomes a relatively easy task, provided that both of the following are true:

1. The QA team has control over the source documents for the specifications.
2. The requirements and tests are tightly coupled.

1.3 The tight coupling of requirements and tests

The concept of tight coupling of requirements and tests is not new. Many other QA teams have explored the advantages of having requirements where tests are easily traceable to their associated requirements and vice versa.

Traceability of the tests and requirements is only a part of the concept of tight coupling, however. To take advantage of traceability, it becomes necessary for the test writer to be able to quickly find the requirement for a given test. In the case where a requirement is being changed or deleted, it becomes necessary for the requirement writer to quickly locate and then change or delete the associated test.

1.4 Giving implementation engineers the timely response they need

It is very difficult to schedule discoveries. Since discovery is an integral part of diagnosing the source of bugs, it follows that it is difficult to schedule bug fixes. In many instances, the discovery of the bug is only a part of the problem: creative solutions are often needed to fix the bug.

From a QA standpoint, scheduling QA runs is less of a problem because they tend to be quantifiable tasks.

When scheduling projects, the verification portion of the project is almost always scheduled as a fixed-length task, and it is always the most stressful: magazine advertisements are bought, conferences are planned, and manufacturing is gearing up for production runs. Software schedule slips at the end of projects are the *most* reprehensible, since they disrupt *everyone's* schedule.

Given that the QA task is the easiest to schedule and manage, it becomes most prudent to make the QA task as fast and as efficient as possible. Every minute saved in the QA process is a minute that can be given to the design engineering team for bug isolation and fixes.

1.5 Changing the test harness for the test writers

In many instances, new requirements for the product can place new requirements on the test harness. It is necessary, therefore, to provide the test writers with the ability to define new test operations in a rather cavalier manner.[1]

Making the test harness as flexible as possible to accommodate the test writers is, therefore, a derived requirement.

[1] I once read an album cover for a Jazz album that referred to an improvisation as the TMIUATGA technique: ''They're Making It Up As They Go Along.'' This concept seems to fit here rather nicely.

2. The design of a quick-turnaround test system

Designing a quick-turnaround test harness is not as difficult as one might expect: our test harness was assembled and made functional in a matter of weeks by two engineers.

Such a harness need not be expensive, either. Our test harness was assembled using some inexpensive, simple, off-the-shelf tools.

We managed to assemble this cheap, effective test harness by looking at the problem *carefully* and from a slightly different perspective than usual.

2.1 The problem of changing requirements

The problem of fast response to changing requirements becomes a problem in fast manipulation of source documents for the *Software Requirements Specification* (SRS). The SRS needs to be easily accessible and easily modified by the persons responsible for the upkeep of this document.

Under ideal conditions, the SRS is a repository of all external requirements for the system software.

2.1.1 Tight coupling of requirements and tests

Some of problems with changing requirements are that:

1. The changed requirement is passed to team members in an informal fashion, and is never recorded in the SRS. This usually happens in later stages of the project when the SRS is perceived as having outlived its usefulness.
2. The test for the changed requirement is never updated. In the case of a dropped requirement, sometimes the test is not removed from the test suites. In the case of an added requirement, sometimes the appropriate test is never added.

Any suitable test harness design will provide for the inherent tight coupling between the written requirements and the test for that requirement.

2.1.1.1 Making requirements writing a QA responsibility

Traditionally, writing the SRS has been a task that belonged to the design engineering team. After the SRS was written, the QA team was then faced with the problem of isolating the design requirements from this document. Often, these requirements were less than explicit.

It is difficult to find fault with the design engineers for the problems that arise with the SRS, since in many instances the true purpose of this document is less than clear.

Testability is a measure of a requirement's quality. If a requirement isn't testable, there is no point in defining it as a requirement.

The QA group has a vested interest in the SRS, and as such, should be more than happy to take charge of such a document. Design engineering, on the other hand, should be more than happy to relieve itself of this responsibility.[2]

[2] In the early stages of this project, I had engaged a member of the design team in a heated debate on specifications. I had generated a list of documents that were to be written and their specific audiences and purpose. I was told by the designer that if I had such a definite idea as to how the SRS should be written, why didn't I assume the burden of writing it? Of course, I seized the opportunity.

2.1.1.2 Embedding the tests in the requirements document

A simple method for binding the tests to the requirement is to embed the test in the SRS immediately after the requirement. As long as the two can be correctly identified within the document, this provides a simple coupling method that provides the necessary link for finding either the test or the requirement. If you want to find the test for the requirement, look immediately after the requirement. If you want to find the requirement for a particular test, look in front of the test.

By breaking the requirements document up into logical, manageable pieces, it is possible to provide the granularity necessary in order to allow several QA engineers to work on different parts of the document at once.

2.1.1.3 Making it easy to write tests

By using a test description ''language''[3] of our own design, we can provide the QA engineers with a means for writing abstract descriptions of tests. If our test description ''language'' is carefully designed, it will be easy for QA engineers to remember the necessary ''vocabulary.''

In instances where our test ''vocabulary'' is inadequate, the test harness should allow for the fast installation of extensions. In short, it ought to allow QA engineers to ''make it up as they go along'' in instances where the current test ''vocabulary'' fails them.

2.1.1.4 Enforcing the disciplines

To enforce the disciplines required of such a scheme, here we would simply advise: use QA engineers, not design engineers. QA engineers will most likely have more of a vested interest in maintaining the requirements and tests than the design engineers will. There seems to be a tendency among design engineers to abandon documents almost immediately after they are written, since document maintenance seems like ''living in the past.''[4]

Enforcing discipline becomes an oxymoron, of sorts. Discipline either exists or it doesn't, and cannot be ''enforced.''

We have made it as easy as possible to find both the test *and* the requirement: they are always together. If you change one, you change the other. If QA engineers are not inclined to practice good document maintenance procedures, all is probably lost anyway.

2.1.2 Timely turnaround of the QA testing procedure

There are several necessary items for timely turnaround in the QA procedure:

1. Fast evaluation and correction of tests: are they written correctly?
2. Fast extraction of the latest tests: getting the embedded tests out of the current version of the SRS as quickly as possible.
3. Fast execution of the latest tests: getting the tests executed as quickly as possible.
4. Fast evaluation of the test results: getting the test exceptions report generated as quickly as possible.

---

[3] The language at use here is the *FORTH* language with a set of custom operators. As the reader will see later on it provides an almost prose-like ''language'' for writing descriptions of tests for the harness.

[4] Before any design engineers take out any contracts, I'd like to point out that I, myself, am a design engineer. I understand how this works, since I have been in this position many times. Remember, I'm on *your* side: did you read the part where I said the QA team should write the requirements specification?

2.1.2.1 Fast evaluation and correction of tests

QA engineers are humans.[5] Humans make mistakes in interesting ways. It becomes necessary to be able to quickly evaluate the validity of a test in terms of correct use of the test ‘‘language’’ and ‘‘vocabulary.’’ If the test harness is sufficiently fast, it is trivial to check the tests: simply feed the test to the test harness and look at the results.

Adopting this approach places additional emphasis on the need to make the test harness operate as quickly as possible.

2.1.2.2 Fast extraction of the latest tests

We have indicated in prior discussion that the design process becomes more difficult to schedule near the end of the project. The same is true for the design of the test cases. After we check a particular run of the latest tests for test errors, it becomes desirable to generate the next test suite from the corrected SRS as soon as possible.

If we place sufficient emphasis on the efficiency of the test extraction software, we can generate the new test suites in a modest amount of time. This turns out to be reasonably simple to do, provided we choose our text formatter and SRS test ‘‘vocabulary’’ wisely.

By providing keyword delimiters that identify the beginning and end of a requirement and the beginning and end of a test, the test extraction software merely has to scan the file and generate the necessary files based on these delimiters.

2.1.2.3 Fast execution of the latest tests

Fast execution of the latest test suite becomes necessary for quick response to new or changed tests *and* to new versions of the product software. In this case, direct execution of the test cases by an interpreter is, in fact, extremely practical.

We adapted a public domain FORTH interpreter to our purpose.[6] Since the FORTH interpreter was written in C, we were able to install our own operators in the interpreter to drive a set of off-the-shelf boards from an instrumentation company. With the assistance of an electrical engineer, we were able to interface these boards with the printer under test.

By using a *state-stimulus-response* approach to our requirements, we are able to write tests with reasonable ease. Using an approach where the requirements were categorized within the SRS along firmware inputs, we were able to specify the response of the printer to every stimulus across all operational modes. While this is an exhaustive approach, it has the advantage of providing excellent coverage in the specification and has the added benefit of accentuating incomplete areas: the SRS is not finished until a response is defined for *every* stimulus across *all* operational modes.

---

[5] There is obvious room for debate here. A QA team I know of was once accused of using an ‘‘army of monkeys’’ testing strategy by a second-level manager whose own heritage had been called into question more than once. A picture clipped from a newspaper of a sheep with its hoof on a terminal keyboard appeared in the QA area. A hand-written caption under the sheep read: ‘‘We will no longer use an army of monkeys to test software.’’

[6] I snagged this FORTH interpreter off of a *USENET* news group (comp.misc.sources), and had it laying around in a directory when this opportunity presented itself. If you’re trying to justify a *USENET* connection to your manager, point out that opportunity favors the prepared pack-rat.

By using the *state-stimulus-response* approach to specification, our initial guess at the necessary operators was that there would have to be five major categories of operators:

1. Operators to put the printer into a particular state.
2. Operators to provide the printer with a particular stimulus.
3. Operators that monitor the printer for a particular response.
4. Operators that provide elementary utilities, such as a ‘‘deadman timer’’ (sometimes called a ‘‘watchdog timer’’) to keep response operators from waiting forever for a response from a ‘‘dead’’ printer.
5. TMIUATGA[7] Operators: Operators that we never even dreamed of.

Our first guess proved reasonably fruitful: it provided over half of the necessary operators.

Also, by providing logging facilities built into the modified interpreter it became very easy to log the progress of the test. Log files were generated for:

1. The requirements themselves. Requirements were embedded in the test suites and are displayed upon the screen of the test harness console as the test is being run.
2. The results of the tests. Every operator in the test suite left a value on the FORTH stack that indicated the success, failure, or timeout of that operation. At the end of the test, if there was anything other than successes on the stack, the test was marked in this log as FAILED (along with the requirement number that failed).
3. Any input from the diagnostic port of the printer under test. Design engineers sometimes wrote important diagnostic information out an RS-232 port on the machine, which we were more than happy to capture and log for them.
4. Any operator instructions that were issued. In the course of testing a printer, it becomes necessary to inspect the resulting prints. In some cases, the test needs to instruct the test harness operator to write an identifying number on the print and lay it aside for inspection by the test analyst. In other cases, the test harness needs to tell the test harness operator to remove jammed or misfed paper from the machine.
5. The test stream itself.

Each test log contains a sequence number that is incremented *anytime* a message is written to *any* file, a time and date stamp, the identifier of the requirement currently under test and the message itself. This makes it possible to extract, merge and sort any combination of the logs to produce the reports discussed in the next section.

2.1.2.4 Fast evaluation of the test results

Fast evaluation of test results is also a critical element of the fast turnaround requirement for the test harness.

A simple utility is written to search the logs for errors, which have been carefully tagged by the test harness with the words ‘‘FAILED’’ or ‘‘TIMED OUT’’. Since these records contain requirement numbers, records containing information germane to this failure can be extracted from the other logs by requirement number, then sorted by the leading sequence number to produce accurate sequential audit trails as to what occurred during the test itself.

[7] ‘‘They’re Making It Up As They Go Along’’ .... Remember?

2.2 The problem of changing tests

Several problems are posed by changing requirements or tests. Most of these are minor problems, by design. The problem of changing tests is met by the following derived requirements:

1. The test harness needs to be extensible.
2. The tests will have to be evaluated and corrected.

2.2.1 The test harness needs to be extensible

The test harness is indeed extensible. The addition of any new operator simply requires the addition of a small amount of C code, or definition of the new operator using a combination of existing operators in FORTH.

2.2.1.1 New test operators will be required as needs are discovered

There will be new test operators required as the test writers discover new operations that will need to be performed. The approach we used was simple: make up the operators that are needed, then they can be installed by a test technician with a minimum amount of effort.

2.2.1.2 New test operators will have to be tested

The new operators can be tested in the test harness itself. The test harness is designed to take FORTH input from the keyboard as well as from file input. The QA engineer testing the new operator can key in an experimental sequence from the console to generate a test case for the new operator.

2.2.2 New tests will have to be evaluated and corrected

As mentioned previously, because of the quick turnaround from the test harness itself, the best way to evaluate new tests is to simply *run them.*

Fast evaluation of the results provide the QA engineer with the feedback necessary to evaluate the tests in a modest amount of time, usually a matter of an hour or two. Once a suite is corrected, it only needs to be retested if more modifications are made.

2.2.2.1 Test operators will be misspelled

There are two approaches to this problem:

1. Correct the misspelling in the test suite.
2. Add another operator with the misspelling, should the number of misspellings prove too great.

Misspellings usually indicate that an operator name was poorly chosen, usually because of inconsistencies with other operator names. Many times it is better to change the name of the operator than to change the test suite: the misspelling will reappear again and again.

2.2.2.2 Test operators will be misused

Test operators will sometimes be misused. Usually this is because of a miscommunication between the test writer and the test harness implementor. Fortunately, this has happened very rarely on this project. The same approach as the previous section applies here:

1. Correct the misuse in the test suite.
2. Revise the operator to behave according to the way it tends to be used.

In almost every case, it is better to revise the operator to behave the way it tends to be used: usually the test writer has been hopelessly indoctrinated in the misuse of the operator. It may well be that the so-called ‘‘misuse’’ indicates a conceptual problem with the test harness implementor. Never put the cart before the horse: support the test writer, not the test harness implementor. In this case the test writer is the customer: *the customer always comes first.*

3. What is MOTHER?

3.1 Meaning of the Acronym

MOTHER is an acronym for ‘‘Maybee’s Own Test Harness for Evolving Requirements.’’ This name itself evolved from a *pet name* that the test harness earned in its early implementation stages. It took a considerable effort to come up with an name that justified the acronym.

3.2 MOTHER is based on a system composed of tools

MOTHER is hardly a monolithic system. MOTHER consists of a series of simple, but highly-specialized tools strung together with a set of scripts. These tools consist of:

1. Document generation tools.
2. Test suite generation tools.
3. Automated test tools (the MOTHER test harness).
4. Automated test exception report tools.

If the reader is more interested in the actual use of MOTHER rather than the building-blocks, I recommend skipping ahead to the section entitled *Using MOTHER.*

3.3 Document generation tools

3.3.1 The ‘‘ms’’ document formatter

The *ms* document formatter is a macro package for the *troff* document formatter. The *ms* package has macros that support various document formats. (This paper was generated using the *ms* macro package.)

3.3.2 ‘‘ms’’ based macros

The *troff* document formatter provides a macro facility that allows users to define their own macros. Since we place requirements and tests in the same file, we may define macros that delimit each requirement and its associated tests. A completely delimited requirement and test would look like this:

```
.RQ
The ready mode shall cause the FAULT line to be asserted
at the interface when the jam access door is opened.
.RE
.TS
SET-READY-MODE
OPEN JAM-ACCESS-DOOR
?FAULTED TRUE-IS-SUCCESS
.TE
```

The macro `.RQ` delimits the beginning of a requirement, while them macro `.RE` delimits the end. The `.TS` macro delimits the start of a test and the `.TE` macro delimits the end of the test.

These macros are defined with a built-in switch that allows QA engineers to print the requirements specification *with* or *without* tests included. If copies of the SRS are needed for reviewing the tests, the ‘‘*include tests*’’ switch can be set to cause copies of the SRS to be printed with both the requirements and their tests. If, on the other hand, the tests are not required, tremendous

amounts of paper can be saved by turning off the ‘‘*include tests*’’ switch.[8]

3.3.3 Source control (RCS)

Off-the-shelf source control tools were used to control the SRS. We chose the RCS package for no other reasons than: it was already there, we were familiar with it, and it did everything we needed it to do.

Primarily, we needed a facility that would allow us to:

1. Control the ownership of the files to prevent two people from working on the same file at the same time.
2. Merge the changes after two people work on different revisions of the same document at the same time.
3. Annotate revisions of the files with commentary describing the changes made to the document.
4. Review the commentary on the changes made to the various revisions of the document.
5. Review what was *really* changed in the various revisions of the document.
6. Remove *improvements* to the files that proved to be detrimental for various reasons.

3.4 Test suite generation tools

3.4.1 Smoke and mirrors: Shell and PERL scripts

Using the fundamental off-the-shelf building blocks described in the previous sections, a little ‘‘glue’’ is needed to piece the entire system together. Specifically, we need to be able to:

1. Extract the tests from the SRS source files.
2. Isolate tests by category: those with outstanding bugs, those that require human intervention to run, those that can run in an unattended fashion, and those that are only partially written (or not written at all).
3. Generate metrics for the test suites: How many tests fall into each of the previous categories.
4. Monitor the progress of the test harness machinery.
5. Generate the test exception reports from the test logs.

3.4.1.1 Bourne shell is the lowest common denominator

We used the *Bourne Shell* as our ‘‘glue’’ for our system. The reasons for this are elementary, and are part of our ongoing theme: it was already there, we were familiar with it, and it did everything we needed it to do.[9]

[8] The size of the SRS as it exists *without* the tests is downright intimidating. It is a reference book, not literature. Imagine trying to read a large volume of mathematical tables as if it were prose. Anything to reduce its size is desirable, especially when giving a copy to an outside organization *for review*.

[9] The reader is probably more than familiar with the advantages of this philosophy. In many instances the time required to study and analyze new tools, the time required for learning new tools, and the associated expense of new tools is prohibitive. Remember, we are striving for *fast* responses. Loosely coupled tools are highly configurable and allow the engineer access to the inner works: if a tool doesn’t do *exactly* what you need to do, rewire it!

The *Bourne shell* also has the advantage of high proliferation. It's everywhere. This gives a degree of portability, but this is a moot point: portability was not one our aims.

3.4.1.2 PERL: Practical Extraction and Report Language

PERL is a report language that is gaining popularity as a replacement tool for ''*sed*'' and ''*awk.*'' PERL has two advantages as a general purpose tool for manipulating files and generating reports: it's easy to learn, and it has a very wide variety of capabilities.[10]

3.4.2 Shell script: build_test_suite

The shell script `build_test_suite` generates the entire test suite from the SRS and sorts the test files into directories on the basis of the status of the test. The status of the test may be as follows:

1. The test may be a fully automatic test. Such test files are sorted into a directory called `auto`.
2. The test may require intervention from a human operator, such as marking a test print with a test number for later inspection. Such test files are sorted into a directory called `manual`.
3. The test may be partially written. That is, a portion of the test may be waiting for the implementation of an operator that was recently ''*invented.*'' Such test files are sorted into a directory called `partial`.
4. The test may not be written at all. The test suite generator is set up to recognize that a requirement may not have an associated test. In this case, these test files (which consists of the requirement portion and nothing else) are sorted into a directory called `no_test`.

The first step in accomplishing these sorts is to combine the multitude of individual files in the SRS into a monolithic unit. This is accomplished by using a utility called `soelim` that comes with the *troff* package. Since the entire SRS is generated by *troff*, individual chapters are ''included'' into the body of the main file using the include macro '' `.so`'', provided by the *troff* package. The `soelim` utility expands all included files into a single output stream that is redirected to a file, thus achieving a file combining all of the latest individual files of the SRS.

The next step involves extracting the tests from the SRS. This is achieved using a PERL script called `extract_tests`. This script takes the monolithic SRS file as input, and watches for the special macro names `.RQ` (beginning of requirement), `.RE` (end of requirement), `.TS` (beginning of test) and `.TE` (end of test).

Since the `.RQ` macro generates a unique requirement number that gets printed with the requirement header, the `extract_tests` script generates the exact same number that is then used as a file name for the tests. In other words, when the document is printed, the requirement numbered 2.4.6.7.8.2 can be found with its associated test in a file *named* 2.4.6.7.8.2, and will be sorted into its appropriate directory in the next step. In the process of extracting the tests, the delimiter macros `.RQ`, `.RE`, `.TS`, and `.TE` are stripped of their leading periods, thus converting them into specialized FORTH operators that will have significance to the harness described in the next section, *Automated testing tool (MOTHER).*

[10] For an excellent series of articles on PERL, see the *Daemons and Dragons* section of *Unix Review*, Vol. 8, Nos. 5, 6 & 7. Rob Kolstad, the author of these articles, provides a very instructive tour of PERL.

The identification of test files is a simple process:

1. If the file contains a FORTH comment beginning in column one, it is presumed that this comment is substituting for an operation that is yet to be decided on. (This is a general convention used by all test writers in our group.) These tests are then sorted into the directory `partial`.
2. If the test contains a key operator that requires human intervention to complete, the test file is placed in the `manual` directory. A file of these key operators is kept in a predefined file, and is read by the `extract_tests` script .
3. If the file is missing a `TS` operator, it has no written test, and will be sorted into the directory `no_test.`
4. If the file does not fit into any of the previous categories, it is then presumed to be automatic and will be sorted into the directory `auto`.

Once all of the tests have been generated, the utility `sort_tests` sorts the tests into their appropriate directories. This proves to be useful from a document management perspective. For instance, once we find a requirement, say, 2.4.6.7.8.2 in the directory structure, we know its status. If we find 2.4.6.7.8.2 in the `no_test` directory, we know that there is currently no test associated with this requirement. This proves to be a most useful scheme when building the tests themselves, since a simple listing of the `no_test` directory gives us a listing of files we need to write tests for, and a listing of the `partial` directory gives us information about functionality still needed in the test harness.

In the next step, the `build_test_suite` calls a shell script `tag_deferred_tests` to move any tests with outstanding bugs into a deferred state. These deferred bugs are tagged with the suffix ‘‘ `.deferred.<bug-number>`’’ on their file name. (`<bug-number>` is of course the actual bug number from the bug database.) Deferral of the tests is an interesting concept, and a test may be deferred for two reasons:

1. The test may be deferred because of an outstanding bug in the bug database. All bugs are tagged with the number of the requirement that is unfulfilled by the bug itself. If test 2.4.6.7.8.2 has an outstanding bug reported against it, there is no sense in running the test again until the bug is fixed. Indeed, it is one less test exception that the test analyst has to wrestle with, and this facility also helps prevent multiple submissions of the same bug reports.
2. The test may be deferred because of another outstanding bug that is wreaking havoc with the test suites in general. Consider the case in which an outstanding bug makes it impossible to get a reliable status report from the printer. This would make it desirable to remove any tests that rely upon a status report from the test suite. Any test may be deferred by placing two fields in a special file used by the `tag_deferred_tests` script: the number of the test to defer, and the bug number of the bug that is responsible for its deferral.

The next step is building the test suites into logical subsuites that will be run on the test harness as individual runs. This grouping is accomplished by a shell script called `block_suites`. This script simply searches each of the directories mentioned above and concatenates all tests in a particular section together. In other words, we happen to know that all requirements in chapter 2.4.1 are requirements dealing with software protocols. It would make sense, then, to collect all files whose name begins with the string ‘2.4.1’ into a single suite and give it appropriate name like: `swproto.for`. The `.for` extension on the file name in this example alludes to the fact that this is FORTH source that may be directly executed on the test harness described in the next

section.

The final step is to collect any metrics on the test suites in general. These metrics are used primarily to monitor the progress of the test writing effort. The shell script `count_suite` produces a report in a file called `count` in the current directory. Here is an actual example of the report produced early in the test harness development cycle:

```
Requirements that have....

    Automatic tests:            633     (  44.5 % )
    Manual tests:               190     (  13.3 % )
    No tests written:           320     (  22.5 % )
    Partial tests written:      277     (  19.5 % )
    Total requirements:         1420

Total number of tests in all suites:   1430
```

This report is produced by simply counting the number of instances of the `TS` and `RQ` operators in the directories. Keep in mind that a requirement may have more than one test.

3.5 Automated testing tool (MOTHER)

The automated testing tool is the physical manifestation of the MOTHER system, and is the portion of the system most frequently referred to as ‘‘MOTHER’’.

The testing tool consists of several elements:

1. The modified FORTH interpreter.
2. The PC-NFS file system link to the server.
3. The test monitor tool.
4. The automated test exception report tool.

Figure 1 shows how the fundamental elements of the PC computer interface with the system under test.

3.5.1 The FORTH interpreter

At the heart of the automated test harness is a public domain FORTH interpreter, written entirely in ‘‘C’’. We have modified this FORTH interpreter to compile under the Turbo-C++ compiler on a PC computer.

We installed a set of custom operators in the FORTH interpreter to allow us to manipulate a set of off-the-shelf instrumentation boards that allow us to drive the printer firmware inputs, and monitor printer firmware outputs.

The FORTH interpreter also has a set of operators that interface with commercially available RS-232 drivers which allows MOTHER to communicate with the diagnostic interface in the printer.

3.5.1.1 Custom test operators

Custom test operators were constructed to interface the test stream with the test hardware, specifically firmware inputs and outputs. These custom test operators were constructed on a three-tier scheme. The three tier scheme was as follows:

1. Operators could be coded in FORTH, as an aggregate of other FORTH operators.
2. Operators could be coded in C, with interfaces to FORTH level calling styles (i.e. parameters could be passed on the FORTH stack).
3. Operators could be coded in C, with interfaces to other C routines. (i.e. parameters could be passed on the processor stack).

This scheme provided engineers with the ability to interface C with FORTH and FORTH with C. It also allowed engineers to construct prototypes of the operators very quickly, using the FORTH interpreter itself.

Experience indicates that the C interface with FORTH operators was never used: engineers worked from the FORTH level down to the C level, and never in the other direction. This would be in keeping with normal top-down programming techniques.

3.5.1.2 Custom hardware interface

The custom hardware interface was a set of off-the-shelf boards that were interfaced with the printer under test. The overall cost of the off-the-shelf boards for each test harness was approximately $1000. At the time the test system was designed, the firmware environment had already been defined, and the hardware interface required some modification of stock hardware to support the test harness.

One of the requirements of the test harness is that there could be no modification of the system under test that affected the firmware. The reason for this is obvious: **if you are testing a custom version of the firmware, you are not testing the *customer's version* of the firmware.**

The hardware was constructed using a *stimulus/release* approach. This approach provides a mechanism which allows the printers native hardware to assert normal operating conditions to the firmware when not overridden by the test harness. To illustrate this concept, consider the following example:

It is desirable to force certain ink-level conditions in the course of testing a printer. We may want to force ink-levels that are *FULL*, *HALF*, or *EMPTY* at the firmware interface. *However, when we are not providing stimulus to the ink-level inputs of the firmware, we want the ink-levels to register their real values.* This is so that in the course of testing that doesn't exercise the ink-levels as part of the suite, the operator will have some indication of whether additional ink is required for normal operation of the printer. (This is especially true when running a suite of tests that makes a lot of prints.) Therefore, in addition to a hardware interface to the ink-levels that provides the *FULL*, *HALF* and *EMPTY* stimulus, we must also provide a *RELEASE* operation that allows the hardware to detect the normal (true) levels of ink.

All hardware interface operators have a *RELEASE* operation, and all hardware interfaces are *RELEASED* at the end of each test. All interfaces are *RELEASED* at the end of each test, so that it is not necessary for the test writers to clean up at the end of each test: the test writers can simply provide the necessary stimulus, check for the results, and leave the system in its current state. The cleanup is automatic, and is an integral part of the TE operator.

3.5.2 The PC-NFS file system

The PC-NFS file system is used to run tests and store test logs directly in the test suite directories. The test suite directories are generated by the server and the server structures are mounted on the PC computer.

As a consequence, when tests are run, the executable image of the MOTHER automated test system and the source suites are actually on the server. The automated test harness writes its log files directly to the PC-NFS mounted directories, and no files are used from the PC. Source files for the MOTHER automated test harness are compiled on the PC and use PC local directories (as opposed to the PC-NFS directories) for development purposes: licensing considerations dictated that our commercial libraries and packages cannot be shared as freely as source files, so we were obliged to keep the development structures local to the PC machine itself.

3.5.3 Test monitor tool

Since the PC-NFS server has both the test suites *and* the test logs available, it is possible to monitor the progress of the suites by comparing the actual suites with the resultant audit trail (test logs) on the server. A utility on the server, `pct_done`, does this comparison and prints the percentage of the testing that is currently done. By using this facility, the test technician can estimate the amount of time left in the execution of a particular suite.

3.6 Automated test exception report tool

Immediately after a particular suite has executed, an automated test exception report tool may be used to generate exception reports. This tool, `failure_reports`, is a shell script that uses simple utilities to generate the exception reports from the test report logs.

3.6.1 Test report logs

The automated test harness writes several error logs during execution of the test suites:

1. `err.log`: The log file of errors detected by the test harness.
2. `mother.log`: The log file of the executed test operators.
3. `zterm.log`: The log file of data received from the RS-232 diagnostic interface (terminal interface).
4. `instr.log`: The log file of instructions issued to the operator during the run of the test suite, and the operator responses to those instructions.
5. `req.log`: The log file consisting of the actual text of the requirements that were displayed during the test. (Requirements are displayed upon the console while the test for that requirement is actually being performed.)

3.6.1.1 Format of the log files

Every time a record is written to a log file, five essential elements are written:

1. *The sequence number.* This is a number that is incremented every time *any* record is written. This number is the first element written to a file, and is a fixed format integer, occupying six columns in our implementation.
2. *The time stamp.* This is primarily to give the test analyst an idea of relative time frames involved. It is also a fixed format field, consisting of hour, minute, second and millisecond.
3. *The number of the requirement under test.* The custom FORTH operator `RQ` places this information in a global variable so that this information is available to all routines within the test harness.
4. *Tag indicating the file that the record was written to.* This tag is an ‘‘E’’ for `err.log`, ‘‘M’’ for `mother.log`, ‘‘Z’’ for `zterm.log`, ‘‘I’’ for `instr.log`, and ‘‘R’’ for `req.log`.
5. *The message.* The remaining information is simply the content of the message written to the specified file.

The following is a brief excerpt from the `mother.log` file of the *ready mode* test suite that illustrates the use of this format:

```
000005 16:44:06.45                -M- ----Stream opened----
000006 16:44:06.67                -M-  [ 0 ] OK
000007 16:44:06.78                -M- RQ  [ 0 ] OK
000014 16:44:07.44  2.4.3.10.0.0.1  -M- TS  [ 1 ] OK
000015 16:44:07.55  2.4.3.10.0.0.1  -M- SET-READY-MODE  [ 6 ] OK
000018 16:44:09.03  2.4.3.10.0.0.1  -M-   [ 6 ] OK
000019 16:44:09.14  2.4.3.10.0.0.1  -M- FULL BLACK INK-LEVEL  [ 7 ] OK
000020 16:44:09.53  2.4.3.10.0.0.1  -M- LOAD-MEDIA ( set ink levels )  [ 9 ] OK
000023 16:44:25.18  2.4.3.10.0.0.1  -M- FORM-FEED  [ 11 ] OK
000029 16:44:46.88  2.4.3.10.0.0.1  -M-   [ 11 ] OK
000030 16:44:47.04  2.4.3.10.0.0.1  -M- DECIMAL 3 SECONDS SET-DEADMAN-TIMER  [ 12 ] OK

etc.
```

The numbers in brackets indicate the depth of the FORTH stack at the time the execution of a line is completed. The `TE` operator will examine the FORTH stack at the end of a test, and print out the depth at which any ‘‘FAILED’’ or ‘‘TIMED OUT’’ markers have been placed on the stack. This makes it simple to locate which part of the test failed, and it also makes it simple to determine the reason for this failure.

The power of this elaborate format will become apparent in our discussion of the test exception report generator in the next section.

3.6.2 Shell script: failure_reports

The `failure_reports` utility extracts lines from the error log that contain the keywords *FAILED* or *TIMED OUT*. Since every line in the error log is tagged with the requirement number, it can extract the actual requirement number from the tagged line. These requirement numbers can then be sorted and piped through a filter designed to ensure that each requirement number appears only once. (This prevents duplicate exception reports, should a particular requirement have more than one failure entered in the error log.)

At this step in the script, we have a unique list of requirements whose tests have logged at least one failure. The next step is to extract the lines from the other relevant files to produce a coherent picture of the actual sequence of transactions that took place. The experience of the test analysts indicate that to form this coherent picture of the actual failure, we need information from the requirements text log (`req.log`), the error log (`err.log`), the diagnostic RS-232 port log (`zterm.log`), and the test log (`mother.log`).

For each requirement in turn, the script:

1. Uses `grep` to extract all lines pertaining to that requirement from the files indicated.
2. Pipes the `grep` output into `sort`, which sorts by the first field (the sequence number).
3. Pipes the `sort` through `pr` to format an exception report listing conducive to analysis. Thus, each page of the exception report has a heading line indicating requirement number, time of generation, etc.

Here is an example of an exception report listing:[11]

```
Jun  4 12:59 1991  Requirement 2.4.3.10.0.0.2                Page 1

Requirement: 2.4.3.10.0.0.2

000067 16:45:28.07  2.4.3.10.0.0.2  -R- Requirement 2.4.3.10.0.0.2
000068 16:45:28.18  2.4.3.10.0.0.2  -R-
000069 16:45:28.18  2.4.3.10.0.0.2  -R- [Window cleared]
000070 16:45:28.29  2.4.3.10.0.0.2  -R- The Ready mode (Ready--), shall proceed to Ink Load mode
000071 16:45:28.34  2.4.3.10.0.0.2  -R- lever), within 2 seconds when a black ink in preload
000072 16:45:28.45  2.4.3.10.0.0.2  -R- position event occurs and the black ink level is low.
000073 16:45:28.62  2.4.3.10.0.0.2  -M- TS  [ 1 ] OK
000074 16:45:28.78  2.4.3.10.0.0.2  -M- SET-READY-MODE  [ 6 ] OK
000075 16:45:30.10  2.4.3.10.0.0.2  -Z- [Ready--]
000076 16:45:30.21  2.4.3.10.0.0.2  -M-   [ 6 ] OK
000077 16:45:30.32  2.4.3.10.0.0.2  -M- HALF BLACK INK-LEVEL  [ 7 ] OK
000078 16:45:30.71  2.4.3.10.0.0.2  -M- LOAD-MEDIA ( set ink levels )  [ 9 ] OK
000079 16:45:46.19  2.4.3.10.0.0.2  -Z- [Printing--]
000080 16:45:46.25  2.4.3.10.0.0.2  -Z- HH: 98
000081 16:45:46.47  2.4.3.10.0.0.2  -M- FORM-FEED  [ 11 ] OK
000082 16:46:07.62  2.4.3.10.0.0.2  -Z-     654    710    837   4126
000083 16:46:07.73  2.4.3.10.0.0.2  -Z- HE: 709 LE: 837 TE: 4126 L: 3289
000084 16:46:07.84  2.4.3.10.0.0.2  -Z- TEND: 4666 TS: 4682 Rem C: 8
000085 16:46:07.95  2.4.3.10.0.0.2  -Z- Park: 5892
000086 16:46:08.00  2.4.3.10.0.0.2  -Z- [Ready--]
000087 16:46:08.22  2.4.3.10.0.0.2  -M-   [ 11 ] OK
000088 16:46:08.33  2.4.3.10.0.0.2  -M- DECIMAL 3 SECONDS SET-DEADMAN-TIMER  [ 12 ] OK
000089 16:46:08.88  2.4.3.10.0.0.2  -M- READY WAITFOR-FRONT-PANEL-MESSASGE WAITFOR-FRONT-PANEL-MESSASGE?
000090 16:46:09.37  2.4.3.10.0.0.2  -M-   [ 0 ] OK
000091 16:46:09.48  2.4.3.10.0.0.2  -M- ASSERT BLACK INK-IN-PRELOAD-POSITION  [ 1 ] OK
000092 16:46:10.09  2.4.3.10.0.0.2  -M-   [ 1 ] OK
000093 16:46:10.14  2.4.3.10.0.0.2  -Z- BLACK
000094 16:46:10.31  2.4.3.10.0.0.2  -M- DECIMAL 3 SECONDS SET-DEADMAN-TIMER  [ 2 ] OK
000095 16:46:10.69  2.4.3.10.0.0.2  -Z- [Pull ink load lever]
000096 16:46:11.08  2.4.3.10.0.0.2  -M- PULL_LEVER WAITFOR-FRONT-PANEL-MESSASGE WAITFOR-FRONT-PANEL-MESSASGE?
000097 16:46:11.68  2.4.3.10.0.0.2  -M-   [ 0 ] OK
000098 16:46:11.79  2.4.3.10.0.0.2  -M- ?INK-SUPPLY-LOW FALSE-IS-SUCCESS  [ 2 ] OK
000099 16:46:12.34  2.4.3.10.0.0.2  -M- ?INK-LOAD-IN-PROCESS TRUE-IS-SUCCESS  [ 3 ] OK
000100 16:46:12.89  2.4.3.10.0.0.2  -M-   [ 3 ] OK
000101 16:46:13.00  2.4.3.10.0.0.2  -M- ?FAULTED TRUE-IS-SUCCESS  [ 4 ] OK
000102 16:46:14.32  2.4.3.10.0.0.2  -M- TE  [ 0 ] OK
000121 16:46:43.10  2.4.3.10.0.0.2  -E- 2.4.3.10.0.0.2 Test 1 operation 2 FAILED
000122 16:46:45.24  2.4.3.10.0.0.2  -Z-
000123 16:46:45.29  2.4.3.10.0.0.2  -Z- [Ready--]
```

In this particular example, the test failed due to sloppy typing by a test engineer that misspelled *MESSAGE* as *MESSASGE*, indicated on lines 89 and 96.

[11] The author regrets to disclose that this exception report listing has been subjected to a little prudent censorship to prevent confusion. At the end of this test, the printer was reset as a result of the failure. The printer is reset whenever a failure occurs, to ensure that it (the printer) is in a quiescent state before the next test begins. The power-up messages contain abbreviated diagnostic messages that are only coherent to the *illuminati* of the design team. Even I couldn't explain the messages contained therein, hence I am reluctant to print examples that defy a complete explanation.

4. Using MOTHER

MOTHER is designed to be simple to use. The environment is geared toward speed and response time. This section describes a simple method for using MOTHER that highlights its flexibility.

4.1 Generating the Software Requirements Specification

To generate the *Software Requirements Specification* (SRS), one need only do the following things:

1. Choose a template, and setup a skeleton of the SRS document. We chose to organize our document along NASA's SFW-DID-08.[12]
2. Write the requirements, delimited in the source files by the `.RQ` and `.RE` macros.
3. At this stage, one *could* begin writing the tests. May I be so bold as to suggest that it would be prudent to subject the document to review first? This precaution may prevent effort from being wasted by writing tests for requirements that may be changed extensively.
4. Write the tests, delimited in the source files by the `.TS` and `.TE` macros.
5. Review the tests.
6. Run and correct the tests.

The method for generating the documents for the above steps is described in the following sections.

4.1.1 Generating the Software Requirements Document

To generate the SRS, a special switch is provided in the body of the document to turn the ''*print tests*'' on or off. If no tests are written for the first review of the requirements then the switch is not of immediate importance.

For later use, the switch will be set to the ''on'' position to generate copious quantities of documentation, showing the tests *in context* immediately next to their requirements.

4.1.2 Generating an SRS listing for review of requirements

The following is a brief example of the format of the document when generated with requirements alone:

[12] I have never actually *seen* this specification. I ran across the outline in a book on writing specifications, and asked our *Technical Standards* folks to locate it for me. I continued writing from the outline of the standard in the book while the Standards folks began what became a *Quest for the Grail*, so to speak. They couldn't locate anyone at *any* NASA facility who had heard of this. Since our *Technical Standards* folks took this as a matter of professional pride, they wrote to the publisher who forwarded their request to the author. The response from the author was essentially ''*never mind*'': the author indicated simply that the NASA requirement would be dropped from the next revision of the book. I wonder if it ever existed at all? It looked nice, but the truth of the matter is that our SRS may be based on a non-existent standard. Although the standard may be non-existent, the format served us very well.

```
Requirement 2.4.1.2.0.0.0.4

The ready mode (Ready--), shall proceed to the power up mode
(Busy--Self Test), within 1 seconds following the release of
the INPUT PRIME line.

Test Description

        Sequence -      (set ready mode)
                        (assert input prime)
                        (check for power up mode)
        Results -       Mode changed to power up within 1 sec.
```

As can be seen, a summary of the requirement and a simple, straightforward description of the proposed test(s) are produced together. This simple description of the proposed test allows the less technically inclined to see the nature of the requirements testing, and allows opportunity for comment.

4.1.3 Generating an SRS listing for review of tests

Where the SRS is to be reviewed from a QA engineering standpoint, a copy must be generated that includes the actual test itself, as in the following example:

```
Requirement 2.4.1.2.0.0.0.4

The ready mode (Ready--), shall proceed to the power up mode
(Busy--Self Test), within 1 seconds following the release of
the INPUT PRIME line.

Test Description

        Sequence -      (set ready mode)
                        (assert input prime)
                        (check for power up mode)
        Results -       Mode changed to power up within 1 sec.

  Test:

                SET-READY-MODE
                RESET
                DECIMAL 1 SECONDS SET-DEADMAN-TIMER
                WAITFOR-FRONT-PANEL-DISPLAY Busy--Self test     "
                ?SELECT FALSE-IS-SUCCESS
```

The resulting document may be reviewed by test engineers to ensure that the tests do indeed test the indicated requirement.

4.2 Generating the test suites

Test suites are generated by using the `build_test_suite` utility mentioned in the previous section entitled *What is MOTHER?*

4.2.1 Manual generation

The test suite may be generated on demand by creating an empty directory and running the `build_test_suite` utility as mentioned in the previous section. The utility itself generates a test log, and, in our case, with about 1400 to 1500 requirements and their tests, it requires about 55 minutes to an hour to generate on an unloaded VAX 8650.

4.2.2 Automated generation using the ‘‘at’’ utility

We have found it useful to generate a test suite using the `at` utility on our VAX. Every morning at about 1 a.m., when the VAX load factor is low,[13] I run jobs that generate a new set of test suites (in case we need them), and a printer-ready copy of the SRS.

4.2.3 Inspecting the results

The `build_test_suite` utility generates output, which I usually redirect to a file called `build.log`. This audit trail will register any problems or fatal errors that are encountered while building the test suites.

Here is an edited sample of what the log file looks like:

```
Tue Jun 04 1991 (01:04:43 AM) Expanding SRS.  Please wait.
Tue Jun 04 1991 (01:06:02 AM) Extracting tests.  Please wait.
Tue Jun 04 1991 (01:10:15 AM) Moving tests into automatic and manual test directories.  Please wait.
Tue Jun 04 1991 (01:10:22 AM) Moving test files containing partial tests
                                  :
                                  :
                                  :
Tue Jun 04 1991 (01:21:50 AM) Moving test files containing WAITFOR-DRUM-HOME
Tue Jun 04 1991 (01:22:29 AM) Moving test files containing WAITFOR-DRUM-MOTION
Tue Jun 04 1991 (01:23:05 AM) Moving test files containing WAITFOR-MEDIA-SCAN
                                  :
                                  :
                                  :
Tue Jun 04 1991 (01:49:31 AM) Marking tests to be deferred.
Tue Jun 04 1991 (01:49:34 AM) Blocking suites.
Tue Jun 04 1991 (01:51:07 AM) Counting the suites.
Tue Jun 04 1991 (01:51:31 AM) Cleaning up.  Please wait.
Tue Jun 04 1991 (01:51:31 AM) Done.
```

The readers who have read the section entitled *What is MOTHER?* will recognize the messages are from the individual utilities that were used at each step of the `build_test_suite` log.

4.3 Running the test suites

Once the test suites are generated, the running of the test suites is simple. From the PC, one need only change to the directory that contains the test suite that you care to run. Then it is a simple redirection of the file. For example, to run the test suite `ready.for` from the console, the test technician only needs to type:

```
mother < ready.for
```

The automated test harness will then provide the rest of the directions, if any, for the test operator.

4.3.1 Automated tests

Once the test harness has been started, tests that have been sorted into the automated test directory can be run without human intervention.[14] The fully automated tests may be started, and essentially ignored until they run to completion.

[13] This is not to imply it is unused. I have received electronic *hate mail* from folks who are working at 1 a.m. They were angry because my jobs ran the load factor up for about an hour.

[14] Yet another great lie. For printers to operate properly, they require a sufficient amount of ink and paper. When they run out, the test harness will stop, request a refill of the exhausted resource and patiently wait for the operator to comply. (There are two great advantages to computers: they perform exhausting and repetitive tasks without complaining, and they wait patiently for the test engineer to get back from coffee break.)

4.3.2 Manual tests

Upon occasion, it is necessary to ask the human operator to do what the harness cannot do: annotate test prints with required information (you simply can't trust the printer to do the annotation itself), load a special size of paper for the test (such as B-size or a metric-sized paper), or any number of other manual operations.

These tests are sorted into a directory of manual tests, and require close operator attention to run. They are sorted into the manual test directory by means of keywords placed in a special file, as described in the previous section *What is MOTHER?*

4.4 Analyzing the output of the test suites

After the run of a particular test suite, analysis may begin immediately. Bugs from the test suite fall into three main categories:

1. Bugs in the printer firmware.
2. Bugs in the test suite (misspellings, etc.).
3. Bugs in the automated test facility itself, such as bugs in the FORTH operators.

The classification of each bug into its correct category is of vital importance to maintaining a healthy relationship with the design team. This is accomplished by:

1. Generating the test exception reports.
2. Inspecting the test exception reports carefully, and assigning bug classifications.
3. Reproducing bugs to be sure that the classifications are reasonable.
4. Submitting the error reports to the design team.

4.4.1 Generating the test exception reports

The test exception report is generated by using the `failure_report` shell script described in the previous section. It generates printer ready audit trails of the tests that failed.

4.4.2 Inspecting the test exception reports

These exception reports are then inspected by the test analysts. The test analyst is the most important element in the test harness operation. We have discovered that, in many cases, the test suites run faster than the test analyst can analyze the results!

The test analyst is important because often the test analyst can spot mistakes in tests at a glance. Errors in this category are usually misspellings or misuse of operators. These mistakes are often very consistent, or ''systematic''. Often, systematic errors in the test suites can be quickly corrected, and the test suite can be run again in the same QA cycle.

4.4.3 Reproducing bugs and submitting error reports

The next stage is for the test analyst to reproduce the bugs by manually stepping through the test suite to ensure that the exception is not dependent upon abnormal conditions. This is done by simply bringing up MOTHER without redirecting standard input as described previously in the section *Running the test suites*. This allows the test analyst to reproduce the bug by typing the operations manually, and to experiment with the condition in hopes of gaining additional information for the design engineering team.

4.4.4 Submitting error reports

In the final stage of the cycle, the QA test analyst can include the actual text of the test in the bug report, so that the bug database carries the actual *reproducible test* for generating the exception. This provides design engineering with an actual plug-and-play bug generation mechanism that cuts the time required to reproduce the bug.

When regression testing is done, the ‘‘bug closure’’ procedure already has the test in place for testing. This shortens the time for regression testing, as a single suite may be synthesized with tests for all of the supposedly ‘‘fixed’’ bugs that can be tested as a monolithic run, if desired.

5. Experiences with MOTHER

5.1 A test harness of this nature is a complete project itself

When planning to build a test harness, it is important to acknowledge that a test harness is a project itself. Basically, there are two things to consider:

1. Test harnesses need project planning, too.
2. Test harnesses need design evaluation as well.

5.1.1 Test harnesses need project planning, too

One of our major oversights was the time required for the development of the test harness. Seldom is sufficient analysis and time given to the correct scheduling of supplemental programs and harnesses for testing: it is at least as much of an oversight in QA efforts as in design engineering (perhaps more so).

The reader is strongly advised not to oversimplify the problem of a project within a project. There are several essential planning elements to consider when building a test harness. These elements are:

1. Manpower requirements.
2. Schedule requirements.
3. Performance requirements.
4. Equipment requirements.

All of these requirements hinge upon the definition of the test harness itself, forcing us to deal with a true paradox:

> *How do we define the requirements for a test harness whose fundamental requirement is to be flexible in its requirements?*

With no experience base to work from, this was enough of a problem to prohibit any worthwhile estimation. Now that we have the figures, which we will share in the next section, the problem is a little less paradoxical.

5.1.2 Test harnesses need design evaluation as well

We have already stated that the test harness is a project within a project. This implies that, as in the case of a commercial project, you have a simple choice: either you evaluate the design of your product or the customer will. In this case the customer is the combined design engineering and QA groups.

An evaluation phase is almost always a painful one for the design engineers. In this instance, the design engineers and the QA engineers are one and the same. It is important that during the design evaluation phase, a certain amount of visibility of this process is given to the design engineering team. There are two fundamental reasons for this:

1. In the future, the design engineers will be asked to accept the diagnosis provided by the harness. It is imperative, therefore, that the design team have confidence in the design of the test harness.
2. In the future, the design engineers will be also subjected to the criticism inherent in the QA team's evaluation of their code. It is imperative, therefore, that the design team does not feel that they are being required to follow design standards that are more rigorous than those followed by the QA team.

5.1.3 We can learn from the implementation of the test harness as well

Formal development of a test harness provides a unique opportunity for the QA community in a company as a whole: it provides a baseline for estimating the effort of future test harness development.

In this particular effort, the QA team tried to capture as much information as possible. Since this effort was, for the most part, a different approach than what is usually used, it wasn't apparent what metrics should be collected. Fortunately, the metrics we did collect seem to be sufficient to answer most questions about developing and scheduling test harnesses of this genre.

5.2 What worked well?

Many things worked well with our test harness: some things worked much better than expected.

5.2.1 Timely turnaround was achieved

The original QA plan called for *three* test harnesses and *five* people. We ended up with *one* test harness and *three* people, myself included. In spite of this, the target two day turnaround was achieved. If we had had the equipment we originally wanted, we could have exceeded this goal, perhaps providing *one day* turnaround. We needed to run three shifts (around the clock) to accomplish the two day turnaround time, but it *was* accomplished. [15]

We were able to scan tests and provide feedback to the test writers in a matter of hours. The longest test suite was about five hours long.[16] The average test suite run time was two to three hours long. In this respect, we satisfied the requirement that we provide fast feedback to the test writers as to the correctness of their tests.

Additions of new test operators were typically accomplished in a matter of an hour or two per operator. Some new operators were installed as a composition of FORTH operators. In these cases, the addition of new operators was typically accomplished literally in a matter of minutes.

[15] I would not advise anyone to try a single-unit approach as described here. We were required to use a single-unit approach because our *Phaser III Printer* prototypes were expensive and extremely hard to come by. The risk of having a single test harness that may be subject to breakdown at inopportune moments makes a single harness approach less than desirable. Also, running around the clock shifts causes extremely high levels of stress among the test technicians. Our test technicians went to heroic lengths to accomplish the two day turnaround.

[16] This is an absolute lie. The longest test suite was 11 hours long until we fixed it so that it worked as it should. A certain amount of honesty seems in order here. If a test suite took more than a couple of hours to run, we did a close analysis to determine why it took so long. In every instance the inordinately long test suites took so long because of some overlooked or unimplemented functionality in the test harness.

5.2.2 Firmware defects in the product under test were exposed

In the course of installing the test harness and running tests against the prototype *Phaser III Printer*, we systematically exposed latent defects in the existing firmware. In short, because we were using the printer in a manner that was never anticipated, we exposed certain defects in the *Phaser III* that may have rarely been seen in everyday use. In some cases, this blatant ‘‘misuse’’ of the printer may have exposed defects that were lurking in the fringes of the firmware architecture itself.

5.2.3 The test harness was ‘‘accepted’’ by the design team

The acceptance of the test harness by the design team was immediate. Although the test harness was released to the design team at a reasonably late time in the project, the ability of the harness to reproduce failure conditions *delighted* the main design engineer.

5.3 What could have worked better?

As is the case with every project, there were instances that left less than desirable conditions for the evaluation of the test subject. In each of these instances, we needed to provide an alternate mechanism for bridging the gap left by inadequacies in the harness.[17]

5.3.1 Hardware interface should have been designed into product

The hardware interface between the test harness and the product was a wiring nightmare. In this product, the QA team arrived late on the scene, and had no input into the hardware design. It is no surprise, therefore, that the hardware was not designed with firmware testability in mind. Since we wanted to use a *state-stimulus-response* approach, it was necessary that the test harness have access to all *inputs and outputs* used by the firmware. Since access to all firmware input and outputs was necessary, we were forced to actually *disembowel* the printer to gain access to these signals.[18]

5.3.2 Granularity of time domain for test was less than desired

The PC/AT provides a less than ideal time base for our tests. In this instance, we ran our time domain tests in an environment where we had an eighteenth of a second resolution of time events. Our test harness took advantage of the system timer tick, and the test harness code intercepted the timer tick interrupt to set a time base for our *deadman timer* facility. Games could indeed be played with reprogramming the system timer chip and passing a reduced number of interrupts through to the operating system. However, there is another barrier lurking in this scheme: the basic instruction cycle time. There is a great deal of potential for marrying the test harness with a programmable logic analyzer for high speed time-domain measurements. This approach is more promising for solving time-domain issues.

5.3.3 Number of test operators was large and difficult to remember

If you make it possible for people to write a new test operator at a moment’s notice, you will find that people will write new test operators at a moment’s notice.

---

[17] This is the real meat of the experience from our perspective. This section is an explicit list of what we will improve upon the next time.

[18] This has its advantages: no one asked to borrow a printer that looked like it was completely disassembled.

The number of test operators was extremely large and soon became somewhat cumbersome. It is doubtful that *anyone* was completely conversant in the entire test description language that resulted. The test writers developed their own test operators that were prose-like and very descriptive, while the test analysts developed their own shorthand that made the same operator easy to type. For instance, the operator `GET-ERRORS-REPORT` was shortened to `GER` by test analysts, yielding two operators that did the exact same thing. While the functionality of the first is readily apparent in its verbose name, the cryptic shorthand notation is most usable by the ''two-fingered'' test analyst who is trying to reproduce the failure of a somewhat lengthy test. Anyone who uses the approach outlined in this paper may want to ensure that their test analysts are capable of touch-typing.

5.3.4 Could have used more time to train test writers

We certainly could have used more time to train the test writers. (In fact having actual test *writers* would have been an advantage: we only had a test *writer*.) The time required to train the test writers would have been spent primarily on providing the writing team with better documentation dealing with:

1. A list of operators grouped by functionality.
2. A list of operators with FORTH's so-called *stack pictures*: a list of what needed to be on the stack *before* the call, and a list of what was left *after* the call.
3. A short paper on the philosophy of the test harness.
4. A few real examples of various types of tests.

These four simple items could have saved many hours for our test writer.

5.3.5 The test procedure was horribly tedious

If you make a test harness that even the simple-minded can operate, then you should hire the simple minded to operate it. The use of the technically literate to perform this extremely mundane task was an inhumane torture. The actual procedure of testing was horribly tedious.[19]

5.4 A test harness can have lasting value

5.4.1 A demonstrated method

We now have a demonstrated method for integrating test and specifications while maintaining flexibility. This is of particular interest to project managers, since it provides for the flexibility of the entire product. This strategy provides for an extremely late binding time for the specification, and this is of the utmost importance for a project such as the *Phaser III*, where simple changes to a single element can ripple through the entire system.

[19] I'm bragging here: It's one of my fundamental beliefs that when engineering gets exciting, it gets exciting in a *bad way*. It is, consequently, a measure of success that this testing process was boring: no ugly surprises. This test harness performed over 1,400 tests with clockwork regularity and consistency. It would be a crime against humanity to require a human to perform the 1,400+ tests with the same accuracy and consistency. Our test technicians listened to radio and played computer games during the periods when the fully automated tests were running: they only had to check on the harness every five minutes or so to ensure that everything was running correctly. In most automated test suites, the *Phaser III* would make enough noise loading paper that it was clear that the system was operating correctly, and only extended periods of silence from the test harness aroused suspicion.

5.4.2 It can only get better

We have provided an analysis of the entire effort showing the areas for improvement, and describing exactly *what* can be done to improve these areas. None of the areas impacted the quality of the product, but rather these areas were primarily associated with making the entire test harness more useful to the customers: the design engineering and QA groups.

5.4.3 A springboard for the next harness

For future products, the test harness as it now exists will cut down on the development effort for the next generation of harnesses. Since all improvements were incorporated in the harness as we went along, it seems reasonable to claim that all the server-based tools are 100% reusable. The actual off-the-shelf boards are, of course, reusable. For any new project:

1. The product-specific test operators will need to be changed.
2. The hardware interface should be cleaned up. (Perhaps a single test harness interface connection to the product under test is in order.)
3. The *Software Requirements Specification* (SRS) will need to be written for the new product.
4. New tests will need to be written for the requirements in the new SRS.

In terms of expenses, there are no new capital expenses for the test harness itself: all new expenses are manpower expenses. All hardware and a large portion of off-the-shelf software[20] are reclaimed.

[20] Compilers, shell scripts and the like. This does not include the test operators themselves, although a large number of them may be reclaimed as well.

6. Conclusion

6.1 Presentation of metrics and numbers

6.1.1 Time to develop test harness

Let me begin by giving a feel for the size of the software in the test harness itself (the PC/AT-based software):

```
Number of lines of code:        9260
Number of blank lines:          4656
Number of comments lines:      11983
```

Applying these numbers to Boehms *cocomo* [21] estimation model, we would project a development schedule as follows:

```
Model mode: organic
Model size: intermediate (9260 lines of code)

Total effort:   24.8 man-months       [152 man-hours/man-month]
Total schedule:  8.5 months           [standard calendar months]

Distributions:                Effort        Schedule       Personnel
                           (man-months)     (months)      (on-board)

     Plans and requirements: (06%)  1.5     (11%)  0.9         1.6
   Product design:           (16%)  4.0     (19%)  1.6         2.5
   Programming:              (65%)  16.1    (59%)  5.0         3.2
     Detailed design:        (25%)  6.2
     Code and unit test:     (40%)  9.9
   Integration and test:     (19%)  4.7     (22%)  1.9         2.5

Programmer productivity during code and unit test phase:  932 DSI/month.
[DSI = Delivered Source Instructions]
```

In actuality, this harness was developed in two months by two engineers. Needless to say, we worked twelve-hour days and most weekends. The implications are that there was a productivity gain of 6.2 *over* the standard man-month. We did not leverage off of any new technology, we just worked *very* hard. Certain productivity gains can be directly attributed to the modularity of the test harness itself: developing lots of little operators is much like having lots of little projects. The FORTH interpreter itself consists of 1031 lines of C code, and has been included in the above code counts for the sake of simplicity.

---

[21] Reference: Boehm, Barry W. *Software Engineering Economics*, Prentice-Hall, New Jersey, 1981.

6.1.2 Time to develop tests

This becomes somewhat of a problem, since the development of tests is an ongoing effort: things were still changing near the end.

We have, at last count, in our SRS (including tests):

```
                    Number of lines:             65840
                    Number of ‘‘words’’:        178583
                    Number of characters:      1350606
                    Number of requirements:       1429
                    Number of tests:              1473

          Percentage of tests requiring manual intervention:   7.9%
```

6.1.3 Time to generate test suites

The generation of test suites took 41 minutes on an unloaded VAX 8650. In addition, it took 2.7 megabytes of disk space to generate the test suites (without the associated log files).

6.1.4 Time to run test suites

There were nine sub-suites that could be run independently. The average time for each of these sub-suites was about 3 hours, although the longest took 5 hours. The total time to run all nine automatic test suites was twenty-seven hours. The time required to run the manual suites was somewhat comparable, but extremely variable depending upon operator response times.

6.1.5 Time to evaluate suites

QA test reports indicate that in the beginning, there is roughly a one-to-one correspondence between run times and the amount of time required to analyze the log files, although this diminishes to practically nothing as the end of the project nears.[22] Once again, this is variable. The logs indicate that one should allocate as much time to the analysis of the results as to the actual running of the suites. By running the proper combination of manual and automatic tests, it is possible to keep the test analyst busy running manual tests and then analyzing previous logs while the subsequent automated tests run. If one is fortunate enough to possess more than one implementation of the test harness, it should also be simple to verify the failures on the second harness while keeping the first harness ‘‘well fed’’ with test suites.

6.1.6 Time to correct suites

In almost all instances, the errors encountered in the test suites were simple misspellings of the operators. These can be corrected in a negligible amount of time, within the SRS itself.

Of particular historical significance was the effort involved in converting the test suites (at a very late date) to a ‘‘language independent’’ format. Language independence was required because the front panel messages could be specified in one of several languages: German, Japanese, English, Spanish, French, or Italian. Rather than duplicate each test for each language, we decided to revise the harness to understand front panel messages in any language. The conversion process involved changing the semantics of an operator which watched for front panel display messages. The operator was to be changed from a *postfix* oriented operator that used a character string parameter to a *prefix* oriented operator that used a constant.

[22] This is, of course, because the number of bugs will drop ... hopefully.

In this instance, the conversion was relatively easy to accomplish using a *PERL* script. A caveat however: when doing such an automated conversion, do not assume that the conversion will be complete. We chose a *new and different* name for the revised, prefix oriented operator and it proved to be a wise choice: the results were a *mostly* converted SRS, and the remaining unconverted operators proved to be a simple job for an editor.

6.2 Will we do it again? You bet! However....

Overall, the experience was a positive one. In order to accomplish the same result again, we must have some of the basic elements we had in the initial effort *as well as* some new concessions.

6.2.1 More realistic planning for implementation of test harness

''Now that we are done, we know how long it will take.''

In the strongest sense of the word, this was certainly a high-risk approach in terms of *effort*: The technology itself was known and proven. The extent and success of this project relied entirely upon the QA engineers who tackled the problems. As is the case with unplanned projects, the amount of effort was entirely glossed over. As a result of doing this, we now have an idea of how long a modification of the test harness should take since we know what sections will have to change, and we can get an idea of their relative size by examining the current test harness.

In short, the next time around will be faster -- much, much faster.

6.2.2 Quality assurance group retains control of software requirement specification

In order to achieve any sort of success using this approach, the QA group *must* have control of the SRS. Without control of the actual framework for such a system, we are hopelessly lost from the start. Engineering specifications for testability are of the utmost importance to a quality product.

Quality assurance is the domain and responsibility of *every* engineer, but it is the charter of the QA group to ensure that it is achieved.

6.3 Acknowledgements and Disclaimers

I feel that every paper should contain appropriate acknowledgements and disclaimers.[23]

As the visionary of the *MOTHER* system, [24] I would be negligent if I did not give proper thanks to the people who actually made it work. The dirtiest and most horrific implementation problems were left to two outstanding engineers.

[23] Here are the disclaimers:

UNIX is a trademark of AT&T Bell Laboratories.

Turbo-C++ is a trademark of Borland International, Inc.

PC/AT and/or *Personal Computer AT* are probably trademarks of International Business Machines Corporation.

PC-NFS is a trademark of Sun Microsystems, Inc.

Ethernet is a trademark of Xerox Corporation.

VAX is a trademark of Digital Equipment Corp.

PERL, grep, pr, RCS, ms, sed, awk, Bourne Shell, troff, and just about everything else mentioned in this paper is a product that we *used* and did not develop: we are not claiming credits or rights to these things. All we did was glue things together in an interesting way.

[24] Someone once suggested that a visionary could be defined as ''someone who is most likely hallucinating.''

First, my most profound thanks to Doug Bingham, a fellow Software Engineer who specializes in design. Doug wrote at least *half* of the 1400+ requirements in SRS by himself, and a full 95% of the tests. He not only accomplished this in a prohibitive time frame, he *never* lost his composure when I said ‘‘*Just fabricate something, and we’ll implement it in the test harness.*’’

Also, my heartfelt thanks to our hired gun, Alan Downing, who connected the test harness to the *Phaser III* printer and implemented a good number of the more arcane operators. Alan endured extremely long hours, changing priorities, and an absurd implementation schedule.

Thanks also to Gary Hanson for the *superb* suggestions for revisions to this paper. My thanks to Jan Maybee, Jack Slingerland, and Ross Taylor, who also proofread the final copy of this paper. (I hope I got everything right this time.)

Finally, my most profound thanks to our *electronics team* leader, Howard Goetz, and our project manager, Ron Adams, for their confidence.

Without the significant efforts of these people, you wouldn’t be reading this. Guaranteed.

CONTENTS