Using TCL/TK for an Automatic Test Engine

C. Allen Flick
DSC Communications Corporation
James S. Dixson
Silicon Valley Networks Corporation

Using TCL/TK for an Automatic Test Engine

C. Allen Flick
DSC Communications Corporation
aflick@wss.dsccc.com, http://www.dsccc.com
James S. Dixson
Silicon Valley Networks Corporation
jdixson@svnetworks.com, http://www.svnetworks.com

Abstract

Test Automation Programming Interface with Object Controlled Authentication, TAPIoca, is a Tcl/Tk based automatic test environment. It grew out of the need to construct automatic tests at the system level that could send commands to a System Under Test (SUT), get responses from the SUT, and control signal generation equipment connected to the same SUT. Tcl/Tk was chosen as the basis for this automatic test environment because of the wealth and maturity of its "language" features, as well as its ease of extensibility.

1 Introduction

Automatic testing has been a widely discussed topic and desired goal for over twenty years in the electronics and software industries. Despite this, there is not one test engine, test management tool, or even test standard that meets all the needs, or desires, of every organization that might use it.

There is a plethora of reasons behind this phenomenon, not the least of which is the necessity for rapid turn around in authoring test cases and collecting their associated metrics to satisfy the needs of managerial reporting. This obligation was the main reason that drove us to choose Tcl/Tk as the language on which to base our TAPIoca automatic test environment. Tcl/Tk, and its extensions, gave us what we needed for the test engineers to generate well formed test cases.

The test cases written in the TAPIoca system could be assembled into scripts that allowed the test organizations to track the progress made in the development of the SUT. Using Tcl/Tk as the language basis of TAPIoca made it easy to assemble scripts and suites of scripts at a pace which could track the development of the SUT. This way tests could be ready for execution when the SUT was ready for testing. TAPIoca has also allowed our organization to quickly change test scripts, if the requirements of the product change (as they quite often do).

TAPIoca is still evolving. Our test engineers who use the tool continue to dream up new features they would like incorporated into it. We've always been advocates of the *creeping elegance* style of development, that is, software should be functional and usable before it can be feature rich. Our approach with TAPIoca has been that it should be used in *real* testing as early as possible in its development. This has allowed it to mature earliest in the areas which were most needed in our testing environment.

2 The Quest for the Ideal Tool

Fine cuisine for one is fodder for another. This is also true when trying to define an ideal test tool. Ask anyone with any significant experience in testing and test automation and you will get a long list of desired functionality ranging from the practical to the impossible.

2.1 A Reality Check

For some, automatic testing means only automatic "collecting" of test results, *i.e.*, a system that prompts the user to perform an action, then asks for the result. The automation system is nothing more than a data entry system for the results of a fundamentally manual activity.

For others, automatic testing means telling a piece of software "verify feature 'xyz' on test-bed '123"'. The test software proceeds to magically configure the '123' device, find the requirements of feature 'xyz', construct and elaborate a set of commands and signals to send to '123', send them to '123', then, using some embedded artificial intelligence, produce a conclusion about the feature 'xyz'.

In truth, reality lies somewhere in between these two extremes.

2.2 Good Automated Testing

Automatic test should do a lot more than just ask questions of a tester, but it most certainly will not be able to infer *everything* about what to do to perform a test.

Good automated testing requires that the test tool/environment have many *domain specific* features above and beyond general capabilities. To us, this meant that the tool had to be able to communicate with:

- RS-232 Devices
- GPIB Instrumentation
- Simulated development devices

In addition, we wanted the tool to have several general qualities:

- A short learning curve.
- Ease of GUI generation.
- Device Control Extensibility.
- A *full featured* scripting language.
- A standard, structured output format for all tests.

2.3 An Existing Tool

We were fortunate not to be designing in a vacuum. We already had an in-house developed test tool called Autotest, written approximately 10 years ago to address domain specific requirements. Autotest included a rudimentary scripting language that testers used to program their tests. It had many of the above mentioned general capabilities, and, overall, most were happy with Autotest, but, it had serious drawbacks:

- It ran only on MS-DOS.
- Its' scripting language was not *full featured.*
- Its' code required high maintenance.
- Its' extensibility was difficult
- It lacked a "batch" execution mode.

Much of our early effort focused on porting Autotest to a Unix environment. But, we realized that to do it right we would have to formalize the Autotest language. This was determined to be a significant effort. So much so, that it opened us to the possibility of using other tools or languages instead.

2.4 Commercial Tool Search

At the time Autotest was written there were no commercial products which could fit the bill. Ten years later when rewriting/replacing Autotest became imperative, we were disappointed to learn that little had changed.

The only two significant commercial contenders for the honor of *one true test tool* were LabView/LabWindows from National Instruments and HP-VEE from Hewlett-Packard. Both of them communicate quite well with GPIB-based test boxes and are relatively easy to learn. Several of our colleagues had developed some quite good, though very specific, automated tests in these environments. If these tools could be proven to do the more general domain testing that Autotest could do, then our quest would be over.

However, trials of both of these tools revealed a great deal of work would be required to implement communication with our SUT's. Furthermore, we found that while the graphical programming environments of LabView and HP-VEE are easy and fun to learn, when we tried to implement a representative sample of automated tests, the graphical programming environment became a real burden.

Using the LabWindows 'C' environment was even worse. It would require the testers to code everything in C. While it was certainly possible to invent a set of LabWindows C-API functions to make test authoring easier, this would have been just as much an effort as porting the Autotest communication code (also in 'C') to Unix.

2.5 Our Dilemma

At this time pressure was building to produce something that would be useful in doing real testing, and real soon. Our evaluation concluded that what our testers really wanted was a scripting environment like our internal Autotest tool, but with modern language features and capabilities.

Again, rewriting Autotest would have required far more time than was practical. Extending environments like LabWindows to include send/expect features *and* be easy to use was just as involved.

3 Our Big Discovery

Our first attempt at a redesign was through Perl, because of previous experience. So, Internet newsgroups were searched for extensions, or methods of using Perl, to meet our requirement of send/expect communication. Noting comments therein to Tcl/Tk and Expect, we looked into their newsgroups. There we found postings from some people right in our own backyard, DSC Communications. Then, further investigation into Tcl/Tk and the Expect extension showed us we had found the "treasure" we were hoping for.

Now that we had discovered Tcl/Tk and Expect, the solution to our dilemma was obvious. **Tcl** contained all of the fundamental language features that we were looking for. **Tk** provided us with easy GUI building that we felt we would need. And, furthermore, the **Expect** extension could easily do the send/expect pattern matching that we had implemented in the Autotest tool.

We also realized that we could easily create some wrapper functions around the Expect calls and end up with a language syntax that was not too different from the syntax of Autotest. This was a big win. This meant that both learning the new environment and converting legacy Autotest code to the Expect environment would be very straightforward.

4 The Prototype

The prototype of this Expect-based testing tool was coded and dubbed **TAPI**, for Test Application Programming Interface. It was a command-line driven application which accepted little more than the name of the script to be run and a few options.

It took little more than two weeks to code and test all the included functionality of Autotest, except GPIB instrument control. It became immediately obvious that **TAPI** had inherited some capabilities from Expect that made it far more useful than Autotest could have been. In particular, it was now trivial for a test script to be directed to a *simulated* SUT, rather than a real device. This would allow our testers to develop and debug tests without the need for time on expensive hardware.

TAPI also differed from Autotest in its Object-Oriented approach to devices under test. One of the things which made it so easy to develop tests within Autotest was that a tester simply had to identify a SUT by the port to which it was connected and send messages to it. Responses were automatically captured, formatted, logged, and compared to specified patterns. If the patterns did not match, an error condition was automatically logged. The tester had to do very little overhead coding to communicate with different SUT's, or log results in a standard format. We decided to adapt this model into a fully OO view of devices under test for **TAPI**.

Despite all these advantages, **TAPI** met with some resistance from the test group. The principle complaint was that while the language was far more capable than Autotest, the command-line nature of **TAPI** was intimidating. The Autotest application had a nice menu-driven front-end which made it easy to start/stop tests and monitor output. **TAPI** did not have that easy interface and, therefore, was not as *good* as Autotest despite the improved functionality. Appearances can be everything.

4.1 TAPI gets GUI

Using Tk, we were quickly able to implement an Autotest look alike interface for **TAPI**. The coding of the basic interface took about a week and had most of the same features of the Autotest interface. Around the same time, it also became known to us that a certain Redmond, WA, based software company had copyrighted the name **TAPI**, so a new name was in order. We decided on the name TAPIoca , which meant internally to us "a Gooey (GUI) version of TAPI", but officially came to be known as "Test Application Programming Interface with Object Controlled Authentication", to reflect the OO nature of the test scripting extensions.

TAPIoca was, and still is, a big hit. It is being used extensively by both our original test group as well as several other organizations in the company. The OO model for devices under test has allowed it to be used successfully in software development groups as well as test organizations.

4.2 GPIB Extension

We also implemented an extension to the basic Tcl interpreter to support communication with GPIB instruments. This turned out to be much simpler than we expected.

The extension capabilities of Tcl really shined here. All we did was create a set of "wrapper-functions" around the desired GPIB calls in the NI-488.2M library and link them to the interpreter.

Once it was done, it was *done*. Now, it's just another set of primitive functions that we use to construct higher level operations.

5 The TAPIoca Architecture

Like Caesar's Gaul, the architecture for the TAPIoca system is composed of three parts:

- TAPIoca API Function Set
- A set of standard **Test Objects**
- A Graphical User Interface

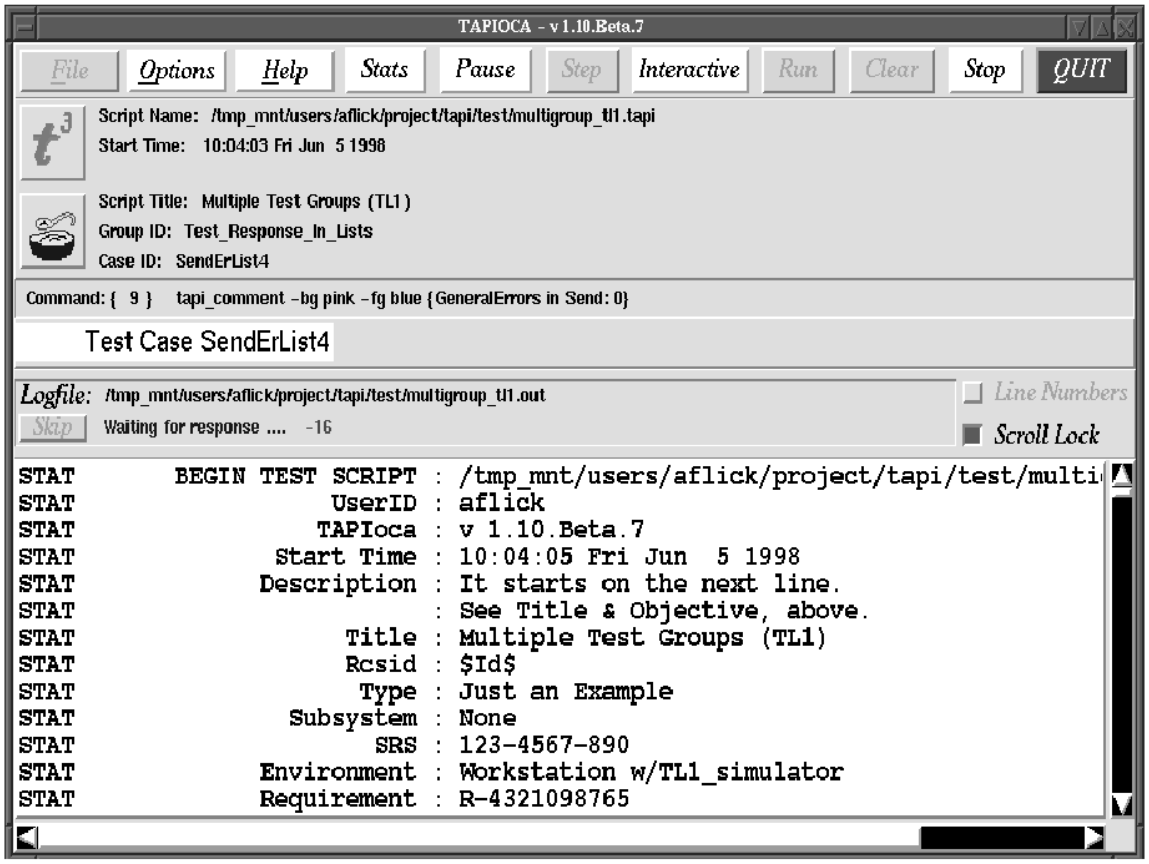
5.1 The TAPIoca API

The TAPIoca API, simply stated, is a set of procedures that implement functions in TAPIoca that are used within all test scripts. These functions provide the testers with the test structure, and overhead, that manage the following:

- Test Group blocking & metric collection.
- Test Case blocking & metric collection.
- Setting Test Group/Case Results
- Adding User Defined Statistics
- Changing Execution Options
- Changing GUI Options
- Creating a Pseudo-Random Number
- User Defined Comments to Output Log
- Displaying User Info to GUI
- Pausing for User Defined Time

Now, some of these, like the *User Defined Time* may appear redundant with core Tcl commands, but in our test environment these are simple extensions that do a little more than their Tcl counterpart.

Take the *User Defined Time* again. Very similar to the Tcl **after** command, but we needed to add to it a display to the GUI that shows the "count down" of the wait time. This is a legacy feature from Autotest that lets the user know the system is NOT dead, but it's actually waiting for something. A similar "count down" is used after a command is sent to the SUT while TAPIoca awaits a response from the SUT. Again, appearances can be everything.



```
Script Name:  /tmp_mnt/users/aflick/project/tapi/test/multigroup_tl1.tapi
Start Time:   10:04:03 Fri Jun  5 1998

Script Title:  Multiple Test Groups (TL1)
Group ID:  Test_Response_In_Lists
Case ID:  SendErList4

Command: {  9 }    tapi_comment -bg pink -fg blue {GeneralErrors in Send: 0}

Test Case SendErList4

Logfile:  /tmp_mnt/users/aflick/project/tapi/test/multigroup_tl1.out
Waiting for response ....   -16

STAT       BEGIN TEST SCRIPT : /tmp_mnt/users/aflick/project/tapi/test/multi
STAT                  UserID : aflick
STAT                 TAPIoca : v 1.10.Beta.7
STAT              Start Time : 10:04:05 Fri Jun  5 1998
STAT             Description : It starts on the next line.
STAT                         : See Title & Objective, above.
STAT                   Title : Multiple Test Groups (TL1)
STAT                   Rcsid : $Id$
STAT                    Type : Just an Example
STAT               Subsystem : None
STAT                     SRS : 123-4567-890
STAT             Environment : Workstation w/TL1_simulator
STAT             Requirement : R-4321098765
```

Figure 1: TAPIoca GUI

5.1.1 API Structure

There is a standard format for all our TAPIoca API and internal **private** procedures. This was determined to be the best approach because testers, in their test scripts, can also "extend" Tcl by writing any procedures they may need. So, we came up with a standardized format and naming convention with which our testers were familiar.

The basic format of our procedures is as follows:

```
tapi_<command> ?args?
        or
tprv_<command> ?args?
```

Therefore, within our environment, any procedure starting "tapi_" is a TAPIoca API procedure, and any procedure starting "tprv_" comes from within the TAPIoca core. Testers are forbidden, by our code inspection process, to create procedures that begin with either of these prefixes.

5.2 A Graphical Interface

To appease our former users of Autotest and to create a test tool that would appeal to most new users, we created a Graphical User Interface.

One of our "unpublished" requirements was that we wanted all of the test scripts written in our replacement system for Autotest to be able to be executed in a "batch", or "headless", environment.

A "batch" mode implicitly means that the test environment cannot require a GUI for normal operation. This goal was foremost our minds when we began designing an architecture for the GUI. For TAPIoca we decided to literally "wrap" the GUI code around the **TAPI** core.

We wanted to preserve a "batch" mode in order to support scheduled execution of test on a remote platform. Previous experience has taught us it is much more difficult to schedule an automated test on a remote platform if that test requires a display to print information on.

We utilized Tk to create the GUI shown in Figure 1. We have our control bar across the top where our menu buttons and tool buttons reside. As shown, the state of each button is controlled during execution of a test script. The lower half of the GUI is what we call our "logview" window. This window displays, in user definable fonts and colors, the output of the test script's execution just as it is written into an output log file. We let the users define the colors and fonts therein simply as a method of making the GUI more acceptable to a wide range of users. Did we mention that "Appearances are everything"?

Controlling what is written into the logview is what we call our *File Viewer*. The *File Viewer* also allows the user to bring up a previously created log file and view it with the user's defined colors and fonts. The *File Viewer* also serves to limit the amount of memory that the logview consumes. We had a problem with limited memory when executing test scripts that generated extremely large output logs. So, by limiting the number of "lines" kept in memory, we minimize the problem for software, and memory consumption becomes a system usage problem.

The remainder of the upper portion of the GUI displays pertinent information for the tester. A quick glance will tell if the test script is operating correctly, and where execution is within the test script.

As shown, we list the active Test Script, Test Group, and Test Case. We also display the current TAPIoca command being executed and its relative position within the test script.

The design was to make the GUI as comprehensive as possible, but at the same time keep it simple and uncluttered. There is always somebody who wants "one more counter" or a "new button" to perform some very application-specific function. We fight vigorously against anything which involves GUI-specific functionality in the **TAPI** core. It is very important to us to defend the "headless" abilities of TAPIoca to run without a GUI interface.

5.3 The Test Objects

The "real guts" of what makes TAPIoca into a valuable test tool is what we call **Test Objects**.

Originally written before [incr Tcl] was available to us, we imitated the OO paradigm in TAPIoca by utilizing some of the "magic" features of Tcl. One of which is Tcl's native feature of allowing a script to define a procedure "on the fly".

To understand what we did, take a close look at the following TAPIoca command:

```
tapi_new <type> <name> <method> <entity>
```

The components of the `tapi_new` command are as follows:

type Declares this object to be a legal TAPIoca type.

name A user defined name used to refer to this object throughout a test script.

method The action to perform on the <entity>. It must match the associated kind of <entity>.

- **-open** Indicates the following <entity> is a *device* that needs "opening".
- **-cmd** Indicates the following <entity> is a program that needs execution.

entity The name of the *device* to be opened, or the program to be executed.

Shown below is a more realistic looking `tapi_new` command that shows the script defining a `GPIB` type object, to be referred to as `hp4532` throughout the test script, and we want device number 8 opened for the interface. In the context of GPIB, this device number refers to the **listen address** of the instrument to which communication is required.

```
tapi_new gpib hp4532 -open /dev8
```

Following a definition of this type in a test script, a procedure has been created by the name of `hp4532`, in our example. Later in the test script, communication with this device is then done by the command `hp4532`, followed by one of various "methods" that is appropriate with this <`type`> of object. Such as issuing the command:

```
hp4532 -send "range7" ?expected-response?
```

If the `?expected-response?` is given, the actual response from the instrument is compared with the `?expected-response?` and flags are set accordingly, enabling the user's script to determine future action, if any, based upon the comparison's result.

There are several other common *methods* that each object type contains. And, some of our object types have `methods` that are specific to that type. Examples of other `methods` are:

-read Just read the device, or instrument.

-config Configure the device parameters (*e.g.* baud rate, parity, etc.).

-login Login to the device under test.

6 TAPIoca at work

There are many interesting things that, internally, TAPIoca does that we could emphasize, but, to be brief, we'll only look at a couple of them.

6.1 TAPIoca scripts

Figure 2 illustrates a typical test configuration. A workstation on the lab network is running TAPIoca, and is connected to the SUT via a RS-232 interface. This workstation is also connected to test equipment via a GPIB bus. The GPIB bus could be either an Sbus interface card internal to the workstation, as shown here, or a bus controller tied directly to the network with its own IP address.

In a TAPIoca test script, the tester could initialize the SUT object and the two GPIB test box objects as follows:

```
tapi_new tl1 msa -open /dev/ttya
tapi_new gpib hp4532a -open /dev6
tapi_new gpib hp4532b -open /dev8
```

Then, in this scenario, SUT setup commands could be issued via the "msa" object command. Likewise, commands to the GPIB test boxes could be accomplished via the "hp4532a" and "hp4532b" object commands. So, a typical test case scenario for a setup like this might follow these steps:

1. Issue SUT setup commands via "msa"
2. Issue GPIB setup commands via "hp4532a"
3. Issue GPIB setup commands via "hp4532b"
4. Read SUT status via "msa"
5. Change SUT setup via "msa"
6. Read SUT status via "msa"
7. Change GPIB setup via "hp4532a"
8. Read SUT status via "msa"
9. Change GPIB setup via "hp4532b"
10. Read SUT status via "msa"

A snippet of actual test code implementing a simple test case is given in Figure 3.

6.2 Test Result Logs

The bulk of output logs generated by TAPIoca are created by the Test Objects themselves. Therefore, when implementing a new Test Object in TAPIoca a great deal of thought is given to the output to be generated.

Each Test Object is responsible for logging all of its relevant activity to the TAPIoca log file. This way testers are not burdened with reporting the details as part of their scripting efforts. We wanted to make the test coding effort as "test logic" oriented as possible, off loading as much overhead and reporting tasks onto the TAPIoca system.

This approach in requiring Test Objects and the TAPIoca API to generate the log output rather than the test scripts has its greatest benefit in ensuring conformity. No matter how poorly coded the test

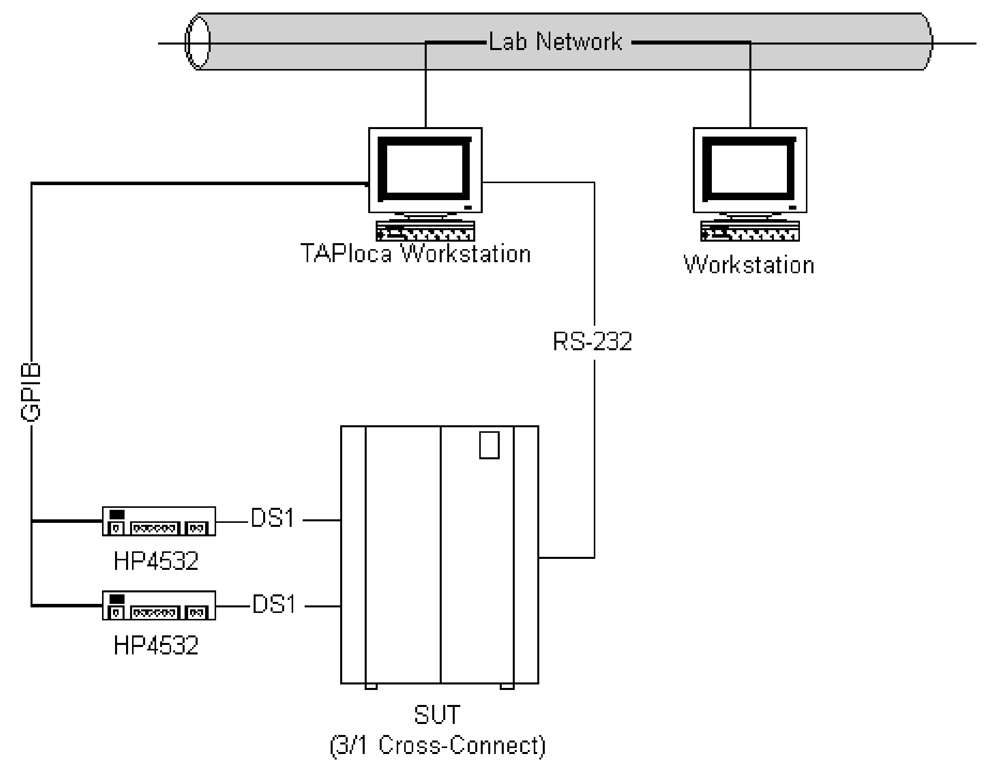


Figure 2: Typical Lab Network Configuration

script is, the output log generated will always contain a complete record of the steps applied to the SUT. This abstraction also allows the output format to change globally without having to change the logic in test scripts.

Figure 4 shows the output generated from the snippet of test code given in Figure 3.

6.3 Integration with Other Tools

The OO abstraction provided by the TAPIoca architecture has permitted TAPIoca to be enhanced to take advantage of newer technologies available. One of the most important of these is TAPIoca's integration with the TestExpert test management system.

One of the features of the TAPIoca environment that was inherited from the desire to work like the Autotest tool is generating a great deal of structured output automatically. The reason for this in Autotest was to support the parsing that output and extracting test metrics automatically. This was never realized in the Autotest world because maintenance of the tool was so high that test tool development had no time for new projects.

Much of the time spent maintaining Autotest internals was freed up after the adoption of TAPIoca. This allowed test tool development to focus on other problems like the management of tests and the collection of metrics.

We decided that the best test management tool would be one which would automatically parse/distill the results of TAPIoca test and insert them into a SQL database. Reports could then be generated from that database.

We were all ready to start development of this system when we discovered the commercial tool TestExpert. TestExpert is a test management system from Silicon Valley Networks that did just what we wanted. It would execute your tests and import the results into a SQL database. It even included a set of canned reports to get us started with report generation. The only problem with TestExpert was that it would only import files in the Test Environment Toolkit journal format into its database.

```
tapi_test_case Number213 {
    tapi_comment "Connect testport 288 to monitor mode"

    msa -send "ttst mon to 0108024 tp 288" \
"M ..:..:.. ..,.. . TTST MON .......,0108024 TP 288 2 LN MSG:<cr><lf>
TRSP TRB SG T CGA.0. TLA R COMPL<cr><lf>"

    tapi_fail_case -miscompare msa "Testport 288 unable to go to monitor mode"
}
```

Figure 3: TAPIoca Code

After some investigation, we realized that the TET format was not much different from the structured output that TAPIoca generated. Taking advantage of our OO architecture, as well as the TET API code provided with the TestExpert product, we were able to get TAPIoca to generate TET journal output natively without any changes to the test scripts themselves.

We can now run TAPIoca test scripts under the control of TestExpert and take advantage of TestExpert features like remote and scheduled execution.

7 The Future

The TAPIoca architecture is based upon our own internally developed OO support code written in Tcl. Going forward, we want to transition this code to the OO model of [**incr Tcl**].

Some work in this area has already been done, but more needs to be accomplished, and we must be careful in doing so, to keep existing test scripts executable.

This fact, as well as the lack of a *released* [**incr Tcl**] for the latest version of Tcl/Tk (8.0), is keeping us from upgrading to 8.0, but we plan to do so as soon as possible.

We also would like to port TAPIoca to the Windows NT environment. One of the requirements of our industry often involves the testing and certification of devices at the customer site. By porting to the NT environment we could more readily take advantage of the lower cost and higher availability of Windows-based laptops.

Last, but not least, is evangelism. We continue to promote TAPIoca internally. Several groups outside our product organization have adopted TAPIoca as their principle test environment. The extensible capabilities of both TAPIoca and Tcl/Tk has spawned several other test groups to write their own TAPIoca test objects.

8 Acknowledgments

When both of us were at DSC Communications, we worked in the same organization. Because of that, our acknowledgment list is a common one. We deeply convey a hearty *Thank You* to all those in this list, both for their role in development and support of TAPIoca within DSC Communications.

Walt Mamed : Our team leader who kept us on track when we wanted to pursue tangential paths of development.

Cheri Snapp : The original manager of our tools group who allowed us the latitude of development we needed, when we needed it. She has championed this test tool for some time and continues to do so in her current position in another division of DSC Communications.

Mark Scafidi : Test manager in another product line within DSC who showed great patience and trust in allowing us to demonstrate that TAPIoca could do **Tk GUI** testing where other products had not met his needs.

And, a special *Thank You* goes to **Jim Edmunds**, our Vice President of product development who provided the funding and significant moral support for

```
CO      =====================================================
STAT         BEGIN TEST CASE : Number213
STAT             Description : Setup odd split mode of fifth set of 12
STAT                         : two-way mapped testports.
STAT              Start Time : 13:14:59 Fri Jun  5 1998
CO      -----------------------------------------------------
CO      Connect testport 288 to monitor mode
SND(msa)    ttst mon to 0108024 tp 288<cr>
RCV(msa),R  M 01:15:08 01,00 4 TTST MON 108024 TP 288 F FAIL DNY<cr><lf>
RCV(msa),R  <si>
ERROR   ******************************************************
ERROR   Miscompare Error
ERROR   Expected:
ERROR   M ..:..:.. ..,.. . TTST MON .......,0108024 TP 288 2 LN MSG:<cr><lf>
ERROR   TRSP TRB SG T CGA.0. TLA R COMPL<cr><lf>
ERROR   ******************************************************
RESULT     TEST CASE SUMMARY : Number213
STAT                  RESULT : FAIL
STAT                  REASON : Testport 288 unable to go to monitor mode
STAT                End Time : 13:15:16 Fri Jun  5 1998
STAT             Miscompares : 1
STAT       END TEST CASE LOG : Number213
CO      =====================================================
```

Figure 4: TAPIoca Output

the project, helping us keep focused on the goal of creating a *practical* test automation tool.

9 Availability

Currently TAPIoca is an internal tool only available inside DSC Communications.

Based on the history of the *Mega Widgets* extension being associated with DSC Communications, we are negotiating with management to release the core of TAPIoca that would not be considered proprietary to DSC Communications. We hope to make that available by year's end.

References

[Expect] Don Libes, *Exploring Expect*, O'Reilly & Associates, Inc. (1995).

[Ousterhout] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishers (1994).

[NI488] *NI-488.2M Software Reference Manual, Sept 1994 Edition*, National Instruments Corporation (1994), Part Number 320351B-01. http://www.natinst.com

[TEAdmin] *TestExpert, The Test Management System: Administrator's Guide*, Silicon Valley Networks (1997), http://www.svnetworks.com