Evaluation of Three ATE Test Environments

Mike Gooding
Larry Cohen
ARGOSystems, Inc.
A subsidiary of the Boeing Company
324 N. Mary Ave, Sunnyvale, CA 94088-3452
Phone (408) 524-1796  Fax (408) 524-2026 E-Mail: mike@gooding.org

Abstract - It is often difficult to assess the positive and negative issues facing the use of a particular software test environment in a given application. Much of the literature is swayed by the use of each environment by a single application. This paper will provide detailed information on ARGOSystems’ evaluation of three popular ATE Test Environments: the Ada Based Environment for Testing (ABET), the TYX PAWS ATLAS test environment and the National Instruments’ LabVIEW graphical test environment. This evaluation was accomplished by comparing the same Test Program measurements in each environment, using the same UUT, Interface Test Adapter, and the same PC-based ATE. As such, the data represents a true apples-to-apples comparison of these environments.

I. INTRODUCTION

ARGOSystems’ Multiple Test Environments project investigated the efficiency and ease of use of three popular test environments: the Ada Based Environment for Test (ABET) used on the F-22 program (Version 5.01), TYX Professional ATLAS Workstation (PAWS), a widely used commercial product (Version 1.1.2) and National Instruments’ LabVIEW, a graphic language commercial product (Version 3.0.1). The effort was completed when 3 functionally identical TPSs, one generated in each environment, were successfully run on the same test station hardware.

To compare the three environments, functionally identical Test Program Sets (TPSs) were generated in each language and run in the corresponding environment. The same Interface Test Adapter (ITA) and target automatic test system (ATS), the Common Manufacturing Test System (CMTS), were used for each. The CMTS is a PC-based ATS that runs both SCO UNIX (for ABET) and DOS Windows (for PAWS and LabVIEW). The 3 TPSs were written from a common English Functional Test Requirement (EFTR). It defines a basic manufacturing acceptance test for the Unit-Under-Test (UUT), an ARGOSystems AS210 Power Supply Module. The test includes resistance, voltage, ripple, timebase frequency and phase jitter measurements. The EFTR defines 86 test steps to be run in a linear fashion. The EFTR does not define a diagnostic tree. Some test step sequences have common elements. For example, the 24 test steps used to test the 24 wires in the IEEE cable are identical except for the interface pins used for each test step.

This paper describes the metrics gathered and derived during the project and presents a summary evaluation of the strengths and weaknesses of each environment. General observations about TPS coding are also included.

II. MAJOR TASKS

As received, each test environment provided a different degree of support for the resources within the CMTS. Device drivers were written for unsupported resources. Table 1 identifies the drivers written for each test environment.

<table>
<thead>
<tr>
<th>ABET Resource</th>
<th>LabVIEW Resource</th>
<th>ATLAS Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 1410 DVM</td>
<td>HP 6050 Electronic Load</td>
<td></td>
</tr>
<tr>
<td>RD 2251 Timer/Counter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WT 1395 Arb. Wave Gen.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RD 1260 Switching</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VXP-1000 Power Supply</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1  Drivers Written by ARGOSystems

A common ITA was designed and built. This included the electrical wiring and the mechanical modifications. The ITA interfaced to 76 unit-under-test pins and 192 CMTS Interconnect Assembly pins.

TPS programmers were chosen that had no previous experience in the 3 test languages. During TPS development, metrics were collected to cover the major functions from initial education through final operation of the TPS.

The ATLAS TPS contained 3,839 lines of text containing approximately 3,500 lines of code equivalent to approximately 1,100 ATLAS statements. The ABET TPS contained 8,530 lines of text containing
approximately 6,000 lines of code equivalent to approximately 600 ABET statements. Lines of text includes comments, blank lines and lines of code. Language statements are typically entered as multiple lines of code for readability. The LabVIEW TPS contained 103 virtual instruments (not including drivers). A LabVIEW virtual instrument is roughly equivalent to a textual module or procedure. LabVIEW has no direct correlation for lines of code or statements.

The ATLAS source files occupy 113,700 bytes. The ABET TPS source files occupy 374,000 bytes. The LabVIEW source files occupy 2,940,000 bytes.

A. Derived Metrics
Table 2 provides relative labor comparisons for incremental tasks derived from the development, not including documentation.

<table>
<thead>
<tr>
<th>ABET</th>
<th>LabVIEW</th>
<th>ATLAS</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>Education</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
<td>Installation</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>8</td>
<td>Support S/W, Drivers</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>TPS Generation</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>9</td>
<td>TPS Debug</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
<td>TPS Gen &amp; Debug Combined</td>
</tr>
</tbody>
</table>

Table 2  Labor Comparison for Incremental Tasks

B. Performance
The 3 environments offer roughly equivalent performance. LabVIEW loaded slowest and ran fastest. Performance parameters are summarized in Table 3.

<table>
<thead>
<tr>
<th>ABET</th>
<th>LabVIEW</th>
<th>ATLAS</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>60</td>
<td>11</td>
<td>TPS Load Time, seconds</td>
</tr>
<tr>
<td>5:46</td>
<td>4:11</td>
<td>5:35</td>
<td>Execution Time, min:sec</td>
</tr>
</tbody>
</table>

Table 3  TPS Performance

III. CONCLUSIONS
All 3 test environments provide functionally equivalent capabilities. All 3 test environments provide equivalent test performance. Each environment has strengths and weaknesses that are discussed in the following paragraphs.

A. Ada Based Environment for Test
ABET provides the most strongly structured and least flexible environment. This may be important for writing very large TPSs that require more than 1 programmer. The test run-time system, although inflexible, is well thought out, pleasing in appearance, and easy to use. The run-time system has 2 log-on permission levels suitng it to manufacturing use. The lowest level is for running tests. The higher level adds the capability to single-step, probe and debug. The run-time system directly supports diagnostic trees and fault isolation. ABET is a complex language to use. Education time is long because the programmer must learn ATLAS syntax rules, ABET language constructs, Ada language and Ada environment rules. In addition, TEDL (Test Equipment Description Language) must be learned to change supporting configuration tables. Assimilating this amount of information requires an experienced and flexible programmer. TPS generation involves time-consuming searches through Ada, ABET and ATLAS manuals trying to find the required information. Because this is largely a "military" language, the manuals are terse and difficult to work with. The best source of information was previously written TPSs that could be examined for applicable techniques. TPS development involves some repetitious entry of data in multiple required documents. Although this has been automated in the Boeing/Lockheed VAX development environment, the PC environment requires hand typing and editing. During program compile and link, Ada produces excellent error messages, allowing syntax errors to be quickly and efficiently corrected. During TPS debug, the run-time system supports probing and bus monitoring making the debug operation very efficient. ABET uses a hierarchy of hardware description data bases that must be generated for the test equipment and Interface Test Adapter using an included TEDL compiler. This structure helps isolate the effects of resource changes and makes the TPS portable to other systems. The ABET environment is moderately robust. Version 5.0 worked well, but did crash and lock up the system in response to several unique hardware conditions.

B. LabVIEW
LabVIEW provides a flexible development environment and a well-structured test run-time system (the Test Toolkit). The environment is well suited to small TPSs generated by a single programmer. The test run-time system is visually pleasing, well engineered and easy to use. It is well suited to the manufacturing environment and has 3 log-on levels. These provide test execution at the lowest log-on level, add hardware debug facilities at the intermediate level and add TPS
debug with modification capabilities at the highest level. This environment comes with source code (the only one to do so) allowing the environment to be customized for special circumstances. The language is well documented, easy to learn and easy to enter. All programming is done in the "G" graphic language as wired pictorial objects in a schematic-like notation. The language environment generates excellent error messages and help information. The test environment provides automatic branching for implementing TPS fault and diagnostic trees. Debug tools make it easy to isolate problems. However, fixing problems that require editing is time-consuming and difficult. The graphic programming environment has no search and replace capability. Making space to add an element on a diagram may require many manual operations to relocate existing elements. Source code files are very large and sometimes will not fit on a floppy disk. Test steps and subroutines are usually grouped in library files that do not allow DOS commands to find creation time and date or file sizes of the individual test steps. This makes change control difficult and good configuration control a challenge. LabVIEW does not require resource configuration tables and typical TPSs communicate directly with the resource drivers. LabVIEW TPSs are the least portable of the 3 environments. The LabVIEW environment is robust. Unique hardware conditions were easily handled by time-outs, allowing the environment to remain functional. The environment has no separate compile and link steps, so program modifications made in the editor can be run immediately. Program loading and initializing is slow compared to the other environments.

C. Professional ATLAS Workstation

PAWS provides a flexible development environment and a moderately structured run-time system. The test run-time system has only one log-on level that provides access to all run and debug capabilities. The run-time display is designed for military use and appears cluttered and confusing by commercial manufacturing standards. Test program interaction with the operator is limited and confusing. There is no facility to pop-up dialog boxes as operator prompts. All prompts appear mingled with the data in the main window. The test progress and result information are not constrained by the test environment but instead appear as programmed in each TPS. This provides flexibility, but also imposes overhead during TPS generation. The result is a lack of standardization to the test operator. PAWS supports a hierarchy of hardware description data bases that must be generated for the test equipment and Interface Test Adapter. This structure helps isolate the effects of resource changes and makes the TPS portable to other systems. The early Windows-based PAWS version reviewed lacked robustness and appeared immature. The run-time system hang in some I/O conditions. The compile system choked on more than 8 modules and can run for days without completing if too many items are entered in the ITA data base. Trouble reports sent to TYX concerning problems were handled well and they are receptive to making corrections in future releases.

IV. DISCUSSION

A. Features

Each environment offers a mix of features. These have been divided into the categories run-time executive, development environment and test program set for presentation in Tables 4, 5, and 6. Additional details are covered in the discussion of each environment in later paragraphs.

The run-time test executive is the portion of the environment present during TPS execution. The test executive allows the test operator to log on, run a TPS, collect test data and then log off. The run-time test executive features of each environment are listed in Table 4.

<table>
<thead>
<tr>
<th>ABET</th>
<th>LabVIEW</th>
<th>ATLAS</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>X</td>
<td></td>
<td>Log-on Levels</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td>Advanced Capability Lockout</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td>Single Test Step Execution</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td>Force Test Step Pass/Fail</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td>Source Code Available to Mod</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td>Built-in Diagnostic Tree Spt</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td>Built-In Bus Monitor</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td>Built-In Debug Probe Capable</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td>Supports Dialog Box Prompts</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td>Fixed-Fmt Test Step Results</td>
</tr>
</tbody>
</table>

Table 4 Run-time Test Executive Characteristics

The development environment is used to code TPSs and supporting data tables. The development environment includes the programming editor and the required compilers, interpreters and linkers required to generate executable TPS object code. The features of each development environment are summarized in Table 5.
Table 5 Development Environment Characteristics

The software portion of the TPS contains the code that must be transported to a run-time test executive to allow execution of a test. Each environment produces TPSs with different characteristics. The characteristics of each are summarized in Table 6.

<table>
<thead>
<tr>
<th>ABET</th>
<th>LabVIEW</th>
<th>ATLAS</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>Text-Based Programming</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>Grapic-Based Programming</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>Choice of Program Editor</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>Separate Compile, Link, Run</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integ Edit, Compile, Link, Run</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linked Dev Tools Available</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>Simulation Run Mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>Automated Help Generation</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>TEDL-Based Config Tables</td>
<td></td>
</tr>
</tbody>
</table>

Table 6 TPS Characteristics

V. PROFESSIONAL ATLAS WORKSTATION

A. Advantages/Capabilities of ATLAS

The main advantage to writing a test program set (TPS) in ATLAS is that the language is widely used by military and airline customers, and they often specify it as the test language they want ATS they buy to support.

From a technical stand-point, TYX PAWS/ATLAS is a general-purpose test language that can read back information from any instrument on a GPIB or VXI bus. The TYX environment allows flexibility in programming test instruments by writing macro drivers. The ATLAS program asks the device data base to provide a test instrument that can measure a desired quantity (such as dc voltage) in a desired range, accuracy and resolution. The program issues a FETCH command to request a numerical result, and the device data base driver issues the commands to an appropriate instrument to return the desired value.

ATLAS is written so that engineers or technicians, who are not ATLAS programmers, can read the source code and understand what the program is doing. Furthermore, test engineers can design programs based on UUT requirements rather than by knowing a specific test system’s capabilities and designing a test plan accordingly. This test philosophy is attractive to the military and other large-volume users.

B. Good Candidate Test Systems

The ATLAS test approach is to issue commands for specific measurements. The ATLAS program assumes an instrument exists which will take the measurement and read back the response. The responses can be further processed in the ATLAS program, but ATLAS depends on the test instruments to do real-time data collection. ATLAS interventions are generally assumed to be slow (non-real-time).

There is extensive capability in ATLAS to modularize the software TPS into procedures. This allows the programmer to write reusable modules and to use the same code many times in the same program without having to copy it throughout the source files.

ATLAS code is UUT oriented. This contrasts with most test systems that are test-instrument oriented. In a test-instrument oriented TPS, the programmer directs the system switch to connect specific UUT pins to specific test instruments. For example, you could direct the switch to connect a power supply output to the system DMM. In ATLAS, you direct that a dc voltage be measured (with optional precision and accuracy) at certain UUT terminals. The compiler decides the switch path and instrument to do the function. In the above example, ATLAS may choose to measure the dc voltage on a digitizing oscilloscope instead of on a DMM. This makes writing code somewhat easier, but may not connect the optimal instrument to make a given measurement.

PAWS/ATLAS programs consist of ATLAS code and three data base files. The ATLAS code follows the IEEE specification for C/ATLAS test language. The data base files allow the ATLAS code to be compiled into a TPS that will run on a specific workstation. It can be expected that any project to prepare a TPS will require writing code for the data bases as well as ATLAS code. PAWS/ATLAS has means of compiling ATLAS code without referring to the switch or ITA data bases. This allows some program debugging to
C. Disadvantages/Limitations of ATLAS

It takes extra effort to force ATLAS to choose a specific instrument to do a measurement. It can be done through the use of REQUIRE statements that uniquely describe the instrument you would like to use, or you can set up a special text file that will force allocation to a specific instrument. This can become an issue if, for example, you wish to measure frequency and there are both a frequency counter and a digitizing oscilloscope in the test system. Both are capable of measuring frequency and the ATLAS program chooses whichever one it feels like using unless the software biases or forces allocation to, say, the frequency counter.

Another function ATLAS has trouble with is reading back the current from a power supply that has built-in current reading capability. ATLAS will tend to choose an instrument with current-measuring capability to fill your request to measure supply current, and the supply itself may not be chosen to do this.

Writing an ATLAS TPS may include writing both ATLAS code and driver macros. ATLAS can be programmed to read back virtually any quantity from the hardware. The hardware, in turn, needs drivers written to answer these requests.

The programmer cannot force a specific switching closure pattern. All switch connections are done automatically by the compiler. If the switching matrix has too many possible choices, the compiler may not have the resources to tabulate all possible combinations and may fail to build the switch data base.

ATLAS screen output is limited to ASCII text sent to the computer screen. There is no capability within ATLAS to generate graphics or alert sounds. Capabilities can be added by calling non-ATLAS modules (NAMs) written in C from within the ATLAS program.

VI. LABVIEW

The LabVIEW graphic program generation/edit environment is labor intensive compared to a textual environment. LabVIEW supports copying any selected patches of schematic. However, it does not have search and replace capability. When 24 identical test steps need a parameter changed, all changes must be made with manual open file, position, select, edit and close file actions. There is no facility for defining macros for repetitive operations. Further, the LabVIEW environment must be used to generate or edit programs. There is no facility to use a favorite schematic drawing package, or other useful tools. The ATLAS and ABET environments do not provide program editors, assuming that a favorite text editor is already available and will be used. Typical programming editors have very powerful copy, replace, search and macro generation capability. Typically, they can record an editing sequence as it is performed by the operator and then repeat it as many times as desired on other code fragments. LabVIEW also took much longer to program simple operations. A typical operation in LabVIEW requires accessing a multiple level menu tree to select first an operator and then a numeric constant block. These must then be moved to position them properly beside each other. Finally the cursor tool must be changed and several movements and clicks performed to wire the two elements together.

As an example, the line "if (a>7) and (b<8) then" took 12 seconds to type into a text editor. The equivalent LabVIEW entry required wiring an AND gate, a switch block, two comparators and 2 numeric constants using 5 wires. This required 1 minute, 20 seconds or roughly 8 times as long. On the other hand, some operations, particularly those involving strings, data clusters (structures) or arrays, are more efficient to program graphically than the corresponding textual statements. The LabVIEW operation to insert a selected line from a multi-line string into the middle of a second string based on a line index number requires wiring a single operator function with a few wires. This may be equivalent to writing a half-dozen lines of textual code and represent a significant time savings.

LabVIEW provides very efficient initial program checkout and debug. If the program is arranged hierarchically, each level can be executed separately without additional programming. The "front panel" controls for that level can be used to set any desired inputs for a test case and the results monitored on the "front panel" indicators. If additional information is needed, additional indicators can be "patched" into the schematic with very little work. Additionally, probe windows can be assigned to any nodes and monitored
independently of the front panel. Text languages typically require that a small test program be written to exercise any lower level code. The test program calls the low level code, supplying the test parameters, and feeding the results to some sort of output routine. Both textual test environments include a code level debugger to support interactive stepping through the code and setting breakpoints. This is a typically a much slower process than the LabVIEW graphical method. LabVIEW animates single step debugging by showing "bubbles" traveling along the wiring of the source diagram. As a "bubble" passes a node it displays the value present. This allows rapid isolation of problems with a minimum of operator action.

The LabVIEW run-time test environment is ideally suited to a manufacturing environment. In the most controlled configuration, an operator can only run full tests and generate reports. Pass/Fail is unambiguously identified for the full test and for individual test steps. Further, the visual presentation and operator interaction is sufficiently constrained that regardless of the TPS design, the operator will be able to run it. The required flexibility for troubleshooting or process investigation is provided by having two additional log on permission levels. Individuals logging in with a password that gives access to the technician level are granted additional capabilities. A technician can run individual test steps in repetitive loops and out of sequence. This capability is typically used to troubleshoot failing units. An individual logging in with a password that gives access to the developer level is granted the most capability. A developer can do anything a technician can and, in addition, can change pass/fail levels and diagnostic tree paths. This capability is useful for temporarily solving or identifying process problems prior to changing the TPS or the process.

LabVIEW is the only environment to provide the source code for the run-time test executive. The test executive can be easily modified with visual enhancements. It took about 4 hours to understand the test executive diagram and then modify it to add custom titles and a green/red pass/fail front panel indicator. It took about 2 hours to modify the screen and file formats for presenting test step data. Functional enhancements would require more time to understand the diagram in detail.

VII. ADA BASED ENVIRONMENT FOR TEST

The ABET environment was developed by Boeing. For the F-22 program, the PC based run-time system was married to a VAX-based development environment that originated with Lockheed Sanders. Special purpose development tools have been developed for the VAX-based environment. For this project, the PC was used as the development environment and the VAX-based tools were not used.

ABET runs under SCO UNIX on a PC. The Boeing ABET package provides the run-time test executive and TEDL compilers for the device model, the configuration model and the adaptation model. This must be augmented with the Alsys Ada Compiler and a text editor to provide complete development capability.

The ABET environment provided all the required drivers for the AS210 test, so the device model was not changed and the TEDL device model compiler was not used. A configuration model, defining the CMTS resources was constructed using a text editor. The configuration model source file was 486 lines long and occupied 20,100 bytes. The TEDL configuration model compiler runs from the command line as a single command. An adaptation model source file, 1,333 lines long, 37,000 bytes, was written to describe the paths through the ITA. The TEDL adaptation model compiler also runs from the command line. The TEDL compilers compiled quickly and produced clear error messages.

The body of the ABET code was established using an early (and unsupported) version of TPS Builder, also developed by Boeing. TPS Builder provides a graphical interface that builds a program structure in response to test entries defining test procedure and test step names. TPS Builder has provisions for defining diagnostic trees, but this facility was not used. TPS Builder generates Ada source files containing basic header comment blocks and procedure begin-end blocks. It also generates several configuration files needed by the run-time system. Unfortunately, the information is generated in the format appropriate to an earlier ABET and hand editing of the results is necessary to reconfigure it for Version 5. Once edited for format, the begin-end blocks are filled with ABET statements to define test step actions.

ABET uses an amalgam of ATLAS constructs and Ada syntax. This provides a unique and fairly readable look to what is basically an ATLAS program. ABET brings tighter syntax control resulting in more robust compiled code. Unfortunately, the Ada definitions of the ATLAS structures are carried in myriad include files (Ada "with"ed files). This burdens the programmer with adding include qualifiers in front of almost every ATLAS token (or learning the language in deep detail to know when this is not necessary). The result is unnecessary complexity and lowered readability of the
source code. Language documentation is terse and lags the actual implementation.

Once the source code is written, it is compiled and linked using the Alsys Ada tools. The Alsys Ada compiler and linker work in a complete Ada development environment that requires that code be kept in libraries that form families. This requires learning and using the Alsys Family Manager and Library Manager tools. Although this helps maintain configuration control and transportability, it represents an additional layer of complexity for small programs. The Ada compiler and linker provide exceptionally clear error messages including suggestions of possible corrections for errors. The Alsys tools all work in a command line style format.

Successfully compiled code is internally consistent but not necessarily correct in reference to test resources or paths. The first time the Ada code is run in the run-time test executive, a linking operation is performed that connects the Ada test step code to the TEDL configuration data bases. This first run generates error messages if references cannot be resolved and linkages established. If successful, the first run generates some additional linkage data base files that are used on succeeding runs. This takes time, giving the initial run a sluggish feel. After the data is saved, later runs proceed rapidly.

The run-time test executive environment provides debug tools for a syntactically correct, linked program with logic or operational problems. A bus monitor window shows and records all bus traffic. An execution probe capability links directly to the Alsys Ada debugger allowing Ada source and assembly language level single-stepping and breakpointing. Although the debugging tools simplify finding errors, correcting them is laborious and time consuming. Once an error is found, the test executive must be exited, the offending code or model text edited, the applicable TEDL or Ada compiler run and then the test executive re-entered. Since the object files have changed, restarting the test executive involves the delays while the link steps are performed and the tables are rebuilt. This entire sequence must be performed for any change, including just changing the pass/fail limits for a test.

The ABET TPS used option 2 as the coding approach. The ATLAS TPS used option 1. The ABET TPS source code was approximately twice as large as the ATLAS, due in part to the different approaches. For contractual programs or large development efforts, the TPS style guide should unambiguously indicate which option (or other approach) is to be utilized before TPS generation begins.

**VIII. OBSERVATIONS ON TPS TEST STEP PHILOSOPHY**

TPS generation can use two fundamentally opposed programming approaches. Option one writes any common code as callable procedures and strives to minimize the code written for each test step. This generates the cleanest test step code and minimizes the amount of typing for entry and editing during debug. Option two makes each test step stand alone, repeating code in each step as required. No user procedure calls are allowed. This increases typing and complicates editing. However, it also isolates the effects of changes. Changes to one test step will not affect the operation of other test steps. The overhead of Option 2 is minimized if the programming editor has good multiple window support with copy and replace capabilities. Graphic languages are at a disadvantage in this area as search and replace operations are not well supported.

The ABET TPS used option 2 as the coding approach. The ATLAS TPS used option 1. The ABET TPS source code was approximately twice as large as the ATLAS, due in part to the different approaches. For contractual programs or large development efforts, the TPS style guide should unambiguously indicate which option (or other approach) is to be utilized before TPS generation begins.

**IX. SUMMARY**

The ARGOSystems Multiple Test Environments project has provided valuable, objective, quantitative and qualitative information that allows deterministic metrics to be used in estimating the application of each environment. It also provides solid rationale for selection of one environment over the other for specific testing applications.

**REFERENCES**