Mission Assurance through Test Program Set Analysis

Yönet A. Eracar, Ph.D.
Teradyne, Inc.
N. Reading, MA 01864
yonet.eracar@teradyne.com

Teresa Lopes
Teradyne, Inc.
N. Reading, MA 01864
teresa.lopes@teradyne.com

Abstract — **This paper provides a brief overview of the challenges of test requirements collection for Test Program Set (TPS) maintenance or re-hosting. The paper specifies a TPS analysis framework to address these issues and provides implementation guidelines using application examples with a focus on digital testing.**

Keywords: ***TPS maintenance, test requirements, TPS re-hosting,***

I. Introduction

Military Automatic Test Equipment (ATE) systems usually have much longer lifetime than their commercial counterparts [1]. Subsequently these systems require decades-long sustainment [2]. Sustainment includes initial provisioning, cataloging or configuration management, inventory management and warehousing, and depot and field level maintenance. Test Program Sets (TPS) are important components of the ATE systems. A TPS includes software (test executive, test program, and instrument driver software), hardware (interface test adapter-ITA, fixture, and specialized instrumentation), and documentation items to customize an ATE system to the test requirements of a unit under test (UUT). This paper focuses on the sustainment of the TPSs.

Two major challenges in sustaining the military ATE systems are configuration management and obsolescence management. Common issues with configuration management are:

- TPS completeness: Archiving test software source files and all their dependencies is an important task, which is sometimes underestimated and poorly implemented. Missing files or failure to archive changes/updates in later revisions prevent the re-build and /or the re-distribution of test programs [3].
- TPS validity: As the ATE systems become more complex and contain more independent pieces, it gets harder to verify the overall functionality of a TPS. Changes/upgrades in underlying test instruments or additional test capabilities in the test program can break the original functionality. Lack or failure to execute an Acceptance Test Procedure (ATP) on a changed TPS will result in an invalid TPS.
- Unused code in a TPS: Unused code clutters up source files for test software and may cause confusion in future software maintenance [4] and hardware requirements collection efforts.

Obsolescence occurs within Department of Defense (DoD) when, (1) an ATE capability becomes degraded due to reduced availability of parts and resources, (2) new technology replaces old, and (3) costs become unaffordable for production, support, and sustainment [2]. When a particular test instrument becomes obsolete, finding a replacement one can be very challenging. Although specification comparison is a good start, it may cause unnecessary constraints/requirements in the search for a replacement. Imagine a legacy voltage buffer instrument that handles up to 500 volts. Is it necessary to require 500 volts max when none of the existing TPSs use more than 220 volts? Should existing TPSs be used as a complimentary data source in addition to the legacy hardware specifications? Yes, the test executive and the test program implicitly contain the information that reflects the test requirements and the necessary specifications for the test hardware. In the absence of a Test Requirements Document (TRD) for a given UUT, test software becomes the main source of TPS requirements collection. Even if a TRD is available, it may not be in sync with the contents and the functionality of the test program due to extensive modifications and poor archiving practices.

When an underlying test instrument is replaced with a new one, the test software needs to be modified to run with the new hardware. The extent of the modifications depends on the original ATE system software architecture and programming language, the organization of the TPS test software, and the software driver for the new hardware. Whether it is a local change or a complete re-hosting of the TPS, a test engineer responsible for modifications is faced with the following issues:

- Understanding the legacy test system and instrument
- Understanding the legacy test program, the programming language and the run-time software
- Understanding the new test system and instrument
- Understanding the new test programming language and the environment
- Understanding the behavioral differences between the legacy and the new hardware

Part of this work was sponsored by Navy under the contract # N68335-08-G-0016.

There are many commercial-of-the-shelf (COTS) applications or point solutions in the market to help with various re-hosting projects. These solutions are provided by new instrument manufacturers as well as third party integration houses. This paper will specify a TPS analysis framework that is based on past examples of TPS re-hosting solutions and will extend them with a set of utilities to address the configuration management and obsolescence management issues discussed above. The paper will start with a list of requirements for such a framework. Then, it will provide implementation guidelines for developing such a framework. Finally, the paper will conclude with a sample implementation of this framework with a focus on digital testing.

II. Requirements for a TPS Analysis Framework

At a minimum, a TPS analysis framework shall meet the following requirements:

- Characterize the hardware capabilities used in the TPSs.
- Report on the utilization of the test instruments.
- Provide analysis of multiple TPSs to aid in the effort estimate for re-hosting from the legacy ATE systems to a new test environment.
- Improve conversion predictability to avoid schedule slips.
- Characterize key technical parameters known to affect re-hosting for each TPS and for an entire collection of TPSs.
- Allow configuration and customization to adapt to future characterization issues.
- Provide mechanisms to leverage past re-hosting experiences.
- Provide automation to minimize user interaction and speed-up processing of large data sets.
- Create reports with a user-configurable level of detail.

III. TPS Analysis Framework

A. General Architecture

The TPS Analysis Framework is a process and a number of tools that support this process. The main components of the TPS Analysis Framework (referred to as the Framework from this point on) are static analysis, dynamic analysis, and query engines. Figure 1 shows the general structure and the major components of a TPS Analysis Framework. The Framework accepts the software source files of the TPS as the input and outputs a set of analysis and query results in a well-defined XML format.

The Static Analysis component parses the TPS source files, extracts the test data, and saves this raw test data in an intermediate format that can be later used by the other components of the framework. The Static Analysis component depends on the programming language used in the TPS software and the Framework can utilize multiple Static Analysis components to support multiple programming languages.

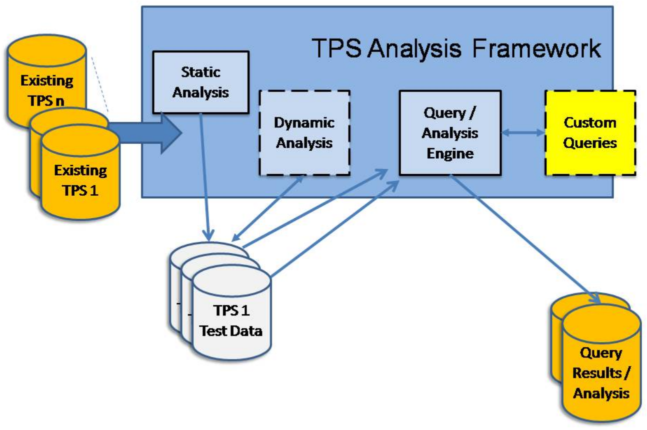


Figure 1 General architecture for TPS Analysis Framework

Dynamic Analysis is an optional component. It uses simulation to detect dynamic changes in the test program based on data returned from the test instruments and UUT behavior. It can determine the values of the variables used in hardware setup, which are not available during Static Analysis. It can also detect unused code sections in the TPS software.

Both the Static and the Dynamic Analysis components can handle only one TPS at a time. The Analysis Engine runs queries across the test data for all the TPSs and generates aggregated reports. The Analysis Engine provides a built-in query editor to create and save queries. It also specifies a programming interface for the users to create more complex, custom queries.

B. Static Analysis

The Static Analysis component contains a parser specific to the programming language used in the test software and a generic test data logger as shown in **Error! Reference source not found.**. As the parser processes the source files, it logs its findings through the test data logger. For example, in an analog test program the logged data may contain an entry for each analog measurement, including the measurement type, range, measured UUT pin name, connection type, the name of the source file, the line number, and the test limits if available. Such data can be very useful to determine the specifications for a replacement test instrument.

Other low level data that may be collected during Static Analysis includes, but not limited to the following:

- Test instrument setup
- Every read and write to the test instruments with setup parameters, and location (file name, line number).

- Measurements from test instruments: type, range, measured UUT pins, connections types, coupling, range, test limits, and location (file name, line number).
- The location (file name and line number) of every logged construct
- UUT pin list, wiring, and switching information.
- Programming language constructs (loops, conditionals), and location (file name, line number)
- Every call to user-defined functions and procedures with input parameters, and location (file name, line number).
- Declarations of global functions, procedures, and variables.
- List of all source files

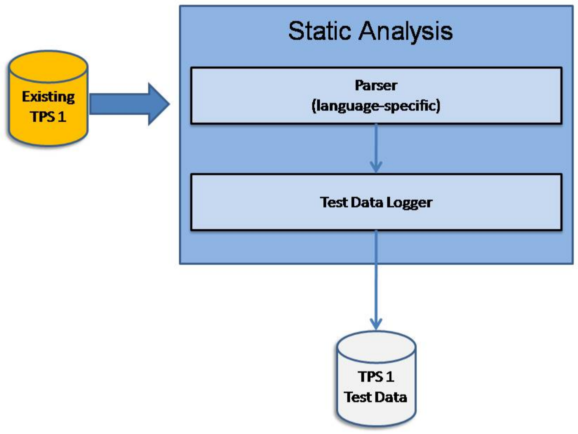


Figure 2 Contents of the Static Analysis component of the Framework

By changing the parser, the Static Analysis component can be adapted to various programming languages. Separation of the test data logger from the parser and standardizing the format of analysis output allows the Framework to support multiple front ends without changing the back end query engine.

The Static Analysis component is sufficient when the test program is deterministic, linear, and the use of variables is limited. Missing files, missing declarations of functions and variables can easily be detected. Test setup parameters, test stimulus and response can be captured. However, the Static Analysis component looks at a static view of the TPS and cannot follow the flow of control or the order of events, if the test program contains conditional processing, i.e. branching based on certain test hardware or UUT condition. These limitations of static analysis necessitate the use of dynamic analysis.

C. *Dynamic Analysis*

The Dynamic Analysis component contains the same parser and the test data logger used in the Static Analysis component, a converter if necessary, a builder utility to create an executable out of the source files, and a simulator as shown in Figure 3.

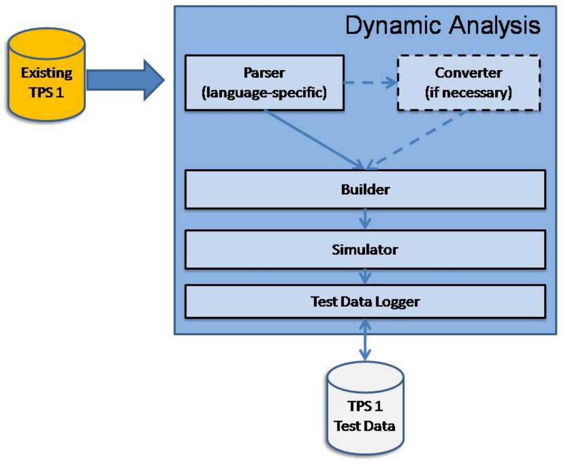


Figure 3 Contents of the Dynamic Analysis component the Framework

A converter utility is necessary, (1) if this is part of a re-hosting effort, (2) if the original programming language does not support simulation and we need to convert the TPS to a different programming language, or (3) if the software drivers for one or more of the test instruments do not support simulation.

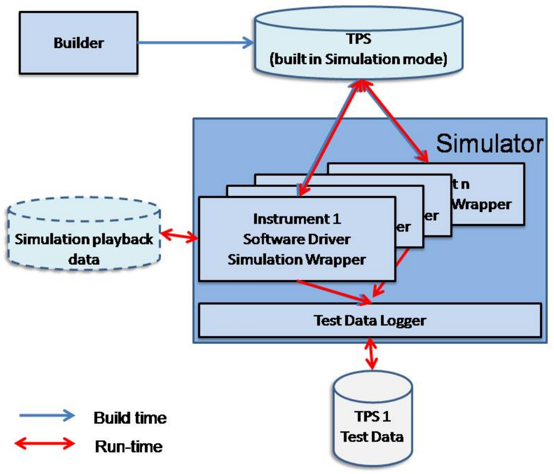


Figure 4 Contents of the Simulator utility in the Framework

The builder utility builds the original (or the converted) source files in simulation mode. The simulator utility runs this simulation program and updates the test data generated during static analysis. To support data logging, either individual instrument drivers should support simulation-time data logging or the converter utility should replace the calls to the instrument drivers with calls to instrument driver simulation wrappers. Here, the instrument driver simulation wrappers are libraries that provide the same entry points as the original driver functions, but support simulation and interface with the test data logger to update the TPS test data as shown in Figure 4.

Dynamic analysis allows the flow of control in the test program to be followed and the values of variables to be determined. Dynamic analysis can also detect unused code sections as well as multiple entries to a hardware setup function with different setup parameters.

One important limitation of simulation is conditional branching based on measurement results from test instruments. Such a limitation can be avoided through the use of simulation playback data where the measurement functions in the instrument driver simulation wrappers return pre-recorded or user-provided values.

D. Analysis Engine

The Analysis Engine component takes the TPS test data generated by the Static and Dynamic Analysis components and one or more user-defined queries as inputs and outputs a set of reports in well defined XML format containing the results of queries (see Figure 5). The Analysis Engine provides a query editor for creating simple queries (referred to as "built-in") and a graphical query results viewer to display the results of the queries in a graphical form i.e. charts view. The Query Engine can handle multiple TPSs at a time. Therefore the query results cover data across all the TPSs. The use of well defined XML format for the query results allows for the user of commercial spreadsheet or database application to further analyze the results.

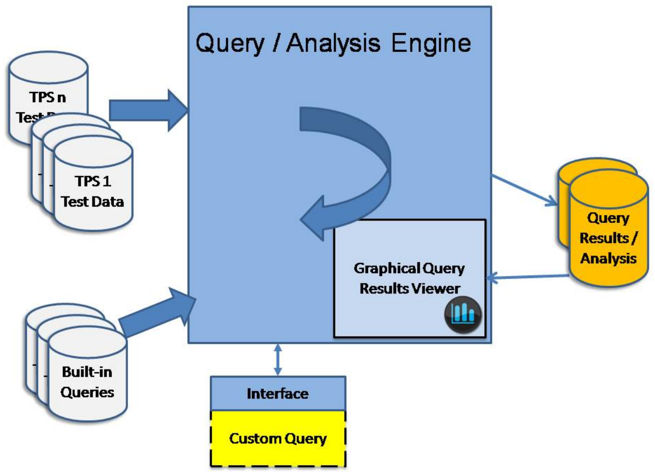


Figure 5 Contents of the Analysis Engine in the Framework

The Analysis Engine defines a simple interface for creating more complex, custom queries. Any user-defined, custom query that implements this interface gets executed together with the built-in queries. The interface passes the location of TPS test data files to the custom query and expects the location of the query results file from the custom query in return.

E. Design for extensibility

Collected data is never enough. The Framework is expected to allow configuration and customization to adapt to future characterization issues. The Framework supports extensibility in two stages:

- Data collection: Separation of parsers form the test data logger and separation of Static / Dynamic analysis from the Analysis Engine and the use intermediate test data allows for changes/upgrades of the data collection mechanisms without affecting the rest of the Framework.
- Data analysis: Support for user-defined processing / custom queries to generate high level data.

F. How does the new framework address the requirements listed above?

Table 1 lists all the requirements and how they are addressed in the Framework.

TABLE I. REQUIREMENTS VS. THE FRAMEWORK

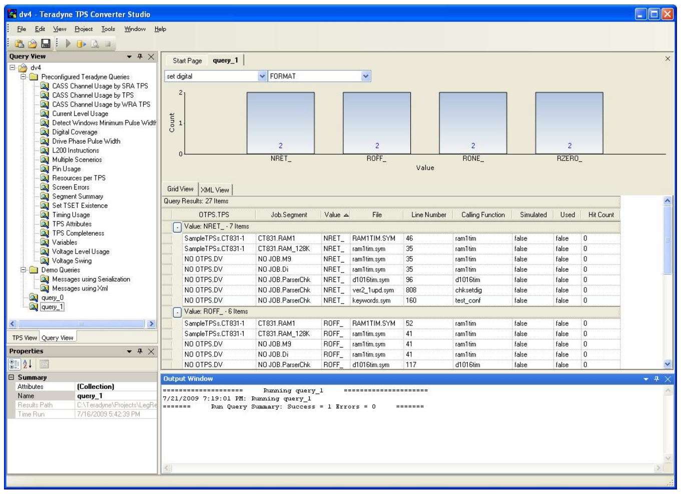
Requirement	**Addressed in**		
	Static Analysis	***Dynamic Analysis***	***Analysis Engine***
Characterize the hardware capability used in the TPSs	Partially	Yes	
Report on the utilization of the test instruments	Partially	Yes	
Provide analysis of multiple TPSs that aids in estimating the effort of re-hosting from the legacy ATE systems to a new test environment	Data collection	Data collection	With a custom query
Improve conversion predictability to avoid schedule slips	Data collection	Data collection	Define a TPS complexity metric using a custom query
Characterize key technical parameters known to affect re-hosting for each TPS and for an entire collection of TPSs	Partially	Yes	
Allow configuration and customization to adapt to future characterization issues	Separation of parsers and test data logger		Use of custom queries
Provide mechanisms to leverage past re-hosting experiences	Data collection	Data collection	Use of custom queries
Provide automation to minimize user interaction and speed-up processing of large data sets			Batch processing of multiple TPSs
Create reports with a user-configurable level of detail			Use of well-defined

Requirement	Addressed in		
	Static Analysis	*Dynamic Analysis*	*Analysis Engine*
			XML format and various XML transformat ion options

IV. Sample Application Using This Framework: CASS TPS Analyzer

The Navy must convert more than 1000 TPSs from Navy's CASS ATE systems [5] to future ATE systems. Most of these TPSs contain some digital content, originally implemented using Teradyne's L200 digital test unit. Given the high cost of rewriting TPSs that can't be automatically transported, the Navy wants to analyze the TPSs for digital functions and hardware settings that are known to cause transport issues. To address this need, Teradyne developed the CASS TPS Analyzer.

The CASS TPS Analyzer uses Teradyne's TPS Converter Studio as the parser and the test data logger in the Static Analysis component. Teradyne's L-Series CShell API was modified to support the requirements of the Dynamic Analysis component of the Framework. The query engine of the CASS TPS Analyzer was architected and developed based on the Query /Analysis Engine component of the Framework (see Figure 6). As part of the project, Teradyne implemented a number of custom queries to look for issues identified in past CASS conversion experiences.



Summary	
Attributes	(Collection)
Name	query_1
Results Path	C:\Teradyne\Projects\LegRe
Time Run	7/16/2009 5:42:39 PM

Query Results: 27 Items

OTPS.TPS	Job.Segment	Value	File	Line Number	Calling Function	Simulated	Used	Hit Count
Value: NRET_ - 7 Items								
SampleTPSs.CT831-1	CT831.RAM1	NRET_	RAM1TIM.SYM	46	ram1tim	false	false	0
SampleTPSs.CT831-1	CT831.RAM_128K	NRET_	ram1tim.sym	35	ram1tim	false	false	0
NO OTPS.DV	NO JOB.M9	NRET_	ram1tim.sym	35	ram1tim	false	false	0
NO OTPS.DV	NO JOB.Di	NRET_	ram1tim.sym	35	ram1tim	false	false	0
NO OTPS.DV	NO JOB.ParserChk	NRET_	d1016tim.sym	96	d1016tim	false	false	0
NO OTPS.DV	NO JOB.ParserChk	NRET_	ver2_1upd.sym	808	chksetdig	false	false	0
NO OTPS.DV	NO JOB.ParserChk	NRET_	keywords.sym	160	test_conf	false	false	0
Value: ROFF_ - 6 Items								
SampleTPSs.CT831-1	CT831.RAM1	ROFF_	RAM1TIM.SYM	52	ram1tim	false	false	0
SampleTPSs.CT831-1	CT831.RAM_128K	ROFF_	ram1tim.sym	41	ram1tim	false	false	0
NO OTPS.DV	NO JOB.M9	ROFF_	ram1tim.sym	41	ram1tim	false	false	0
NO OTPS.DV	NO JOB.Di	ROFF_	ram1tim.sym	41	ram1tim	false	false	0
NO OTPS.DV	NO JOB.ParserChk	ROFF_	d1016tim.sym	117	d1016tim	false	false	0

```
====================     Running query_1     =====================
7/21/2009 7:19:01 PM: Running query_1
=======     Run Query Summary: Success = 1 Errors = 0     =======
```

Figure 6 CASS TPS Analyzer using the Framework

A. *Data collected*

In addition to the data listed in Section III.B, the CASS TPS Analyzer collects the following data via a set of Teradyne-provided custom queries:

- CASS digital channel usage by CASS station type
- Level information: Current and voltage level usage, range, swing, separation, use of undefined level sets
- Timing usage: Frequency detect windows and drive phases minimum and maximum pulse width, use of undefined timing sets
- Used L200 digital functions and parameters, the use of explicit settings vs. relying on hardware defaults
- Use of variables
- TPS completeness: Missing files, function/procedure, and variable definitions.

B. *Issues Found*

In CASS TPSs, Abbreviated Test Language for All Systems (ATLAS) is used as the test programming and flow of control language. ATLAS is a high-level test language, which is independent of test instruments. An ATLAS feature called Non-ATLAS Modules (NAM) is used to extend the functionality of ATLAS when an instrument capability or other programming construct is not available through the ATLAS language. In CASS TPSs, control of Teradyne's Digital Test Unit (DTU) is done using L200 language and runtime. The ATLAS program communicates with the L200 program through a mail boxing scheme implemented in NAMs.

Since the CASS TPS Analyzer only handled the TPS source files written in the L200 programming language and TPSs required both the ATLAS and L200 parts to run, simulation proved to be the most challenging issue. A simple ATLAS parser was implemented to extract the calls to NAMs that communicate back to the L200 programs. A simple ATLAS client application, rather than a full-blown ATLAS simulator, was developed to execute this list of NAM calls extracted with the parser. While this approach worked in some cases, it failed in an expected fashion when the NAMs were passed information stored in variables. Since the creation of a complete ATLAS simulator was not part of this project, a mechanism is provided to allow the user to modify the list of NAM calls and specify input parameters when necessary.

V. Conclusions

In this paper, we started with an overview of the challenges of sustaining military ATE systems and TPSs, with a focus on Configuration Management and Obsolescence Management. Whether it is routine updates or major parts replacement, a thorough analysis of the TPSs is essential to maintain the validity of the TPSs and minimize unnecessary costs. In Section II, we created a minimum list of requirements for a solution that will help with such an analysis effort. In Section III, we specified a TPS Analysis Framework with a complete architecture. We made sure that the specified Framework will be extensible to address future data collection and analysis requirements. We concluded the paper by describing a real-life application, called CASS TPS Analyzer, which was modeled using the TPS Analysis Framework.

.Future work involves investigating better, automated mechanisms to create simulation playback data that will result in more realistic simulation runs.

Acknowledgment

We would like to acknowledge Kevin Lemoi for his contributions.

References

[1] P.A. Buxbaum, “Obsolescecne Management”, Military Logistics Forum, Vol. 3, Issue 5, June 2009.

[2] Defense Acquisition University, “Obsolescecne Management”, https://acc.dau.mil/CommunityBrowser.aspx?id=18073

[3] M. A. Lapham, “Sustaining softwar-intensive systems”, CMU/SEI Technical Note TN-007, May 2006.

[4] D. Mueller, “Software tool for validating and verifying re-hosted legacy software and interface hardware,” Autotestcon, IEEE 2006, pp. 360--365, September 2006. *(references)*

[5] “Consolidated Automated Support System (CASS)”, http://pma260.navair.navy.mil/.