A Framework for Automated System Testing

by

Thayne R. Coffman

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

May 28, 1996

Author .....
Department of Electrical Engineering and Computer Science
May 22, 1996

Certified by ....
Barbara Liskov
pervisor

Accepted by ....
enthaler
Chairman, Department Committee on Graduate Theses

A Framework for Automated System Testing

by

Thayne R. Coffman

Submitted to the
Department of Electrical Engineering and Computer Science.

May 22, 1996

In Partial Fulfillment of the Requirements for the Degrees
of Bachelor of Science in Computer Science and
Engineering and Master of Engineering in Electrical
Engineering and Computer Science

Abstract

The development of the System Test Automation (STA) tool helps fill a gap in combined hardware and software test automation. STA is able to exercise a system's software via command-line or graphical interfaces, and simultaneously control equipment to exercise the system's hardware. Once developed, test sequences can be executed without human assistance. This allows automated regression testing, which has already exposed four defects in the first system it tested at Hughes Network Systems. The STA framework is independent of the system under test, and includes function libraries that can be applied without modification to other products once tests for those product have been coded. By structuring STA as libraries attached to a generic framework, it is hoped that STA will become an example of successful code reuse. Although STA is still in the prototype phase, other divisions in Hughes Network Systems are considering adopting it in their development cycle.

Thesis Supervisor: Barbara Liskov
Title: N.E.C. Professor of Software Science and Engineering

Table of Contents

1 Introduction 6
1.1 Purpose and functionality 6
1.1.1 Improvements to the TES development cycle 6
1.1.2 Flexibility, expandability, and reuse 8
2 Background 10
2.1 Historical quality assurance 10
2.2 Automated testing systems 11
2.2.1 Selection criteria 11
2.2.2 Advantages and disadvantages 13
2.2.3 Examples 14
3 Development environment 19
3.1 Work environment 19
3.2 Testing environment 20
3.3 The TES system 23
4 System Design 25
4.1 Design overview 25
4.2 Operator interface 27
4.2.1 Interactive mode 28
4.2.2 Batch mode 32
4.3 Framework 32
4.3.1 System configuration 32
4.3.2 Controlling sequences, tests, and execution flow 33
4.3.3 Hardware configuration 34
4.4 Sequences and tests 37
4.4.1 Sequences 37
4.4.2 Tests 39
4.4.3 Dynamic alteration of the test sequence 41
4.5 Libraries 41
4.5.1 Overall library structure 41
4.5.2 HP VEE libraries 45
4.5.3 XRunner and HP VEE / XRunner interface libraries 45
4.5.4 Hardware libraries 47
4.5.5 Compiled libraries 53
4.6 Naming and documentation 55
4.6.1 Naming conventions 55
4.6.2 Documentation requirements 58
5 Conclusion and New Directions 62
5.1 Design evaluation 62
5.2 Performance evaluation 66
5.3 New directions 66
5.4 Conclusion 68
Bibliography 70

List of Figures

Figure 3.1 Test development and execution environment 22
Figure 3.2 TES system layout 23
Figure 4.1 System overview 26
Figure 4.2 STA operator interface structure 29
Figure 4.3 Overall STA library structure 42
Figure 4.4 Existing generic libraries 43
Figure 4.5 Hardware library structure 49
Figure 5.1 Existing TES STA directory structure 65
Figure 5.2 Proposed TES STA directory structure 65

List of Tables

Table 4.1 Summary of naming conventions 56

Chapter 1
Introduction

Changes in the required quality of computer systems and changes in the scale of systems currently under development have forced the hardware and software industries to explore new quality assurance methods. Increasing chip densities and new fabrication techniques complicate hardware quality assurance, and larger and more complex software systems complicate software quality assurance. Automated test equipment (ATE), computer assisted software engineering (CASE) tools, and other automation techniques have become the standard rather than the exception for modern quality assurance efforts.

In spite of these developments, there are relatively few tools to automate the testing of combined hardware and software systems. These combined tools can perform tests isolated to hardware, tests isolated to software, and tests that exercise both hardware and software as a system. This thesis documents the initial development of the *System Test Automation* (STA) tool being used at Hughes Network Systems (HNS). The STA system helps fill the void of automated testing tools for combined hardware and software systems.

1.1 Purpose and functionality

1.1.1 Improvements to the TES development cycle

In addition to its interest as an example of a testing tool in an application area which is less automated than other areas of testing, the STA system also provides more pragmatic benefits. The most concrete driving factor in the development of the STA system was the automation of TES regression tests. The STA system was developed in the Telephony Earth Station (TES) group at HNS. The TES product is a satellite telephony network (described in section 3.3), which employs both custom hardware and software. The size of

the TES product prohibits adequate manual regression testing, and the length of the development cycle is growing as a result. Builds of the TES system are done once a week, and manual regression testing is not a realistic option. Once regression tests are automated, there is the opportunity to automate other stages of testing. A higher level of quality can be achieved once tests are coded and automated, and the developers will be less burdened by the testing process.

In order to encourage the use of the STA system by the developers, test design and coding needs to be as easy as possible. If the development of automated tests took more time than manual testing, the system would provide no benefit. Also, developers would be unlikely to use the system if test development was tedious or irritating.

The initial development of the STA system was driven by the goal of automating a subset of the TES release tests. These prototype tests were [HNS1]:

1. Create a TES configuration database using the command line interface to the TES network operator console. The network site has telephone, data, control, and monitoring channels.

2. Configure the remote site for the proper signalling protocol using the operator console GUI.

3. Modify the configuration of one channel at the remote site and verify that the modification occurred.

4. Place a telephone call from one voice channel to the other. This includes initializing external call generators, placing and answering the call, sending test tones over the connection in both directions, and terminating the connection.

5. Create a data connection between the two data channels. This includes defining the connection in the database, initializing external data generators, establishing the connection, sending test data in both directions, determining the bit error rate for the test data, and removing the data connection.

This set of tests exercised the complete functionality desired of the final STA system. The STA system accessed the TES software through both a traditional command-line

interface and a GUI. It controlled and utilized the Ameritec AM8e call generator and the Fireberd 6000 data generator to perform hardware tests. Furthermore, some tests depended on the response of both the TES hardware and software, which satisfied the requirement to support combined hardware and software tests.

The prototype tests also exercise STA functionality that is not directly visible from a definition of the tests.

1. The STA system can complete an entire test run without any operator present (i.e. during non-office hours). This allows the use of physical assets during times they would otherwise be idle.

2. The function libraries required to perform a test are dynamically loaded into and removed from memory as they are needed.

3. Tests are grouped into *sequences*, and these sequences are grouped into test runs.

4. Complete sequences or individual tests can be enabled or disabled at run time by the operator. If no operator is present, information on which tests and sequences to run is loaded from a file.

5. The sequence of tests in a run can be modified by the STA system during execution.

6. Tests are parameterized to allow their customization by an operator at run time. If no operator is present, default parameters are loaded from files.

7. The results of the test run, including a pass/fail status, error code, and text description of the result, are displayed on the screen and logged in a time-stamped file.

1.1.2 Flexibility, expandability, and reuse

The STA system was developed with the intent that it would be reused on different product lines and possibly in different stages of testing, with a minimum of additional effort. The desire for reuse shaped much of how the STA project was viewed and designed. Changes were expected in the functionality of the systems under test (SUTs). Changes were also expected in the types of hardware that STA would need to communicate with. Different groups might use different test equipment, new test equipment might

be introduced into the development cycle, or SUT hardware might be able to communicate with the STA system directly. Changes in the methods used to communicate with hardware were also expected. Finally, changes in the stage of testing being performed by STA would require both its flexibility and expandability.

The STA system was developed as two related but separate projects, the *TES Automated Regression Testing* system and the *System Test Automation Generic System.* The System Test Automation Generic System is what is referred to as the STA system. It contains all the parts of the project that had the potential for reuse in other divisions within the company. The TES Automated Regression Testing system was an instantiation of the generic system for use on the TES product line. Separation of generic and project-specific elements was a major requirement throughout the system, in the hope that the STA system will provide one of the few examples of successful software reuse.

The desire for reuse also motivated the creation of guidelines for adding functionality to the system. These included conventions regulating the structure, documentation, and naming of new STA tests, libraries, and library functions.

Chapter 2
Background

2.1 Historical quality assurance

Quality assurance for both hardware and software can be separated into two main classes: static analysis and dynamic analysis. Each class uses its own tools and methods. The books and articles listed in the references detail different quality assurance methods and tools.

Static analysis methods are those used to validate designs or implementations that do not actually exercise the hardware or software under consideration. The use of theorem-proving algorithms to formally prove that the required behavior of hardware or software logically follows from the given implementation is known as formal verification. Other static analysis techniques analyze implementations to verify structural integrity, check adherence to conventions, or find syntax errors.

Dynamic analysis methods are more commonly known as "testing". They are the central quality assurance methods in use today for both hardware and software. The STA system is designed to help automate dynamic analysis of combined hardware and software systems.

Testing methods involve exercising the system on inputs and checking that output behaviors match predefined expected outputs. Because the input space for both hardware and software testing is too large to test exhaustively, much effort has been given in each realm to finding sets of tests that are good approximations to the complete set of possible inputs.

2.2 Automated testing systems

Automated testing (ATS) systems are the high end of testing tools. They require less human intervention during the testing process than traditional automated test equipment. Thus ATS systems can offer improvements to the testing process unavailable with traditional ATE. The boundary between traditional testing tools and ATS systems is somewhat arbitrary, and I have chosen to require that an automated testing system be primarily controlled by a computer. The interaction between the tester and the test run should be minimized. This section describes a few examples of automated testing systems that are already in use.

2.2.1 Selection criteria

The first criterion any successful ATS system needs to meet is long-term economy. There are four different aspects to this requirement: high utilization, reliability, maintainability, and a long service life [Lon70]. High utilization reduces the average cost an ATS system contributes per unit. ATS systems can offer very high utilization because they need not always have an operator present. This translates to a savings in employee cost. The level of decision-making capability needed to have a testing system run extended test sequences without an operator present is not available in normal ATE, and is best found in computer-driven ATS systems [Cla70]. Reliability limits the time an ATS must be under repair, increasing utilization. Maintainability is an issue because the systems being tested are continually developing. Maintainability is a direct consequence of a well-designed, modular system. A flexible system will be able to run on many generations of systems under development, giving it a long service life.

The interface between the tester and the ATS system is of the utmost importance in an ATS. Many of the benefits gained by employing an ATS system stem from improvements in the interface between the tester and the system under test. The more we eliminate the need for humans to be involved in the execution of tests, the more time and money we save-- "One of the greatest time thieves in the production environment is the interaction of the test operator with the test equipment [Cli85]." The importance of the interface between the tester and the testing system was a major factor in the choice of Hewlett-Packard's Visual Engineering Environment (HP VEE) as the primary language in the STA system.

A properly-designed ATS system is much more flexible and general than normal automated test equipment. ATE for testing hardware is often able to test only one or a few types of devices [Hea81]. An ATS system should be nearly independent of which type of hardware it is testing. Furthermore, it should be independent of whether it is testing software, hardware, or a combination of both. If flexibility is the end, modularity is the means. It must be expected that almost every aspect of the testing environment will change over the lifetime of the testing system. The type of ATE being controlled, the methods used to communicate with that ATE, the functionality and interface of the SUT, and even the phase of testing in which the ATS will be used can all change. All of these changes were considered in the design of the STA system, which will be described in more detail later. Hardware Interface (HIF) and Application Programmer Interface (API) libraries allow changes in the ATE being controlled, Ports libraries allow changes in the methods used to communicate with ATE, software interface changes were isolated in global variables, and new libraries to allow the use of the STA system in other phases of testing are under devel-

opment. In general, the framework/library structure used by STA is a very modular and flexible architecture.

A final requirement for a successful automated testing system is the capability to analyze test results. Bad analysis of good test data is often a source of error in testing. An automated testing system will usually produce much more output than previous manual testing procedures, so the tester is definitely in need of a tool to assist him in the analysis and interpretation of testing results [Lew92].

2.2.2 Advantages and disadvantages

There are a number of advantages to automated testing systems but also a few disadvantages. These factors clearly weigh towards the use of ATS systems. They reduce testing time, reduce operator error, minimize the personnel required for testing, and run more tests than humans could do in the same time. One of the most important advantages of ATS systems is that they allow developers to identify and correct defects earlier in the development cycle through enhanced regression testing. These advantages are enough to justify their use. There are, however, other beneficial side effects from the use of ATS systems.

Automated testing requires test procedures to be coded in a computer-readable language. The explicit formulation of test procedures in a formal language eliminates confusion that could result from stating test procedures in a natural language. Because test procedures, inputs, and outputs are all available on-line, the use of an ATS also makes it easier to keep these documents organized and under configuration management [Deu82].

Another side-benefit of automated testing systems is that new types of tests can be performed. Some tests require simulations of the environment that cannot be done accurately

enough by hand, or they may have timing requirements that a human tester can not consistently meet. Automated testing system can do these tests, and may enable the testers to run whole classes of tests that could not be reliably run before [BL70].

While the advantages of automated testing systems far outweigh their disadvantages, disadvantages still exist. With careful planning and a well-structured system, however, many of them can be avoided. For example, automated test equipment is usually limited in the number and type of devices it can control. It can interact with only one or a few types of hardware. Because of its limited scope, it is not as cost-effective and useful as it could be [Lon70]. Similar to the problem with limited ATE scope, ATS systems have typically required a considerable amount of maintenance because the systems they are testing are constantly changing in functionality and interface [BL70]. The STA system localizes changes in hardware functionality to device-specific libraries, and localizes changes in software interface and functionality to global variables used to define the interface.

One disadvantage that can not be eliminated is the need for testers to be familiar with the language in which the tests must be written [BL70]. While there is no way to eliminate this learning curve, new programming languages with ease-of-use as a goal have taken steps toward reducing it. As was stated before, this was a primary factor in the choice of HP VEE as the main implementation language for STA.

2.2.3 Examples

AVS

The Automated Verification System discussed in [Deu82] was designed exclusively for testing software and was concerned with measuring the coverage of test cases and assisting in the preparation of new test cases. There were five basic functions performed

by the AVS system. The first two functions served a typical static analysis goal-- creating a database and analysis report that could identify problems areas in the code, and identify the control and data structures. The third and fourth steps ran a predefined set of tests on the code and analyzed the results and coverage of those tests. The fifth step is the most interesting, in which AVS automatically generated reports to aid in organizing testing and increasing test set coverage. While none of these steps in isolation was any great feat, their combination in a single system is notable.

ATEX

The ATEX system described in [BL70] is a system used to simulate operator and hardware actions for the U.S. Navy's "Naval Tactical Display System" (NTDS), extract and evaluate console outputs, and produce a listing of the results. The NTDS system is used on U.S. Navy ships to aid the combat center in its command and control activities.

The ATEX system was comprised of three segments: a compiler program, a control program, and a reduction program. The compiler program took previously recorded operator actions and response checks and automatically created computer-coded test scripts from them. The control program executed the scripts' instructions on the NTDS and recorded test results. The reduction program took the raw test output and processed it so it was more useful in test analysis.

One interesting aspect of the ATEX system is that it would test the NTDS while it was on-line. This meant that the operation of the ATEX system had to be as invisible to the crew of the ship as possible.

ELT

The ELT (End of Line Test) system described in [May85] performed analog and digital testing of sub-assemblies. It required an operator, but it could test nine items between each operator intervention. It was capable (through software storage of hardware characteristics) of testing over 260 different types of devices. Relevant in this system is the use of the IEEE-488 bus to control remote hardware. The IEEE-488 bus is a standard in automated test equipment, and many ATEs can be controlled via this bus. While it was not used to control external hardware in the STA system, HP VEE supports communication through the IEEE-488 bus, and extending the capability of STA to include this bus would require only very minor additions to one library.

ABBET

ABBET, "A Broad-Based Environment for Test", is a family of IEEE standards (1226.x) that was developed to increase the levels of automation and information sharing in the testing of hardware and software. The goals of ABBET were to improve test programming languages, develop standards to characterize and control test instruments from software, and promote the reuse of information related to testing between phases in the hardware and software life cycles [Hei95]. The standard is independent of the ATE equipment used, and supported the simulation of equipment when it is not physically present.

ABBET is a very general standard for defining the type of information used in the testing process. The ABBET information model forces testing-related information to be recorded in a way that promotes the maximum reuse across development phases and even across companies [TMR95]. Each dependance a testing component (testing strategy, testing procedure, ATE, testbed configuration, etc.) has on any other testing component is

abstracted away into one or a few modules in the system. Because of this architecture, a testing component can be reused when one of those dependencies changes without explicitly modifying its relationship to any other part of the system. The ABBET architecture is general enough that it can be used for hardware testing, software testing, or both. It is also concrete enough that an instantiation of it has been successfully demonstrated by industry companies [NMH95]. ABBET is based on a general framework and collection of generic and project-specific libraries, just like the STA system.

"ATS"

The last example of an automated testing system is unfortunately named just that, "Automated Test System (ATS)", so it will be referred to as the Lockheed-Boeing (LB) system. It is described in [CW95].

The purpose of the LB system is exactly that of the hardware libraries of STA-- to be able to control different types of external test hardware automatically, and to be able to write test scripts that are independent of which equipment is being used to execute the test. The LB system was able to control over 200 types of ATE. The structure used in the LB system is very similar to that of the STA system, and the issues raised in the development of the LB system were considered and accommodated in the STA system. This is encouraging because the existence of the LB system was not known during the development of the STA system, so the two systems were developed independently.

The LB system's architecture is very similar to the structure of the hardware libraries in STA. It consists of an instrument-independent layer of services, on top of an instrument-dependent layer of services, on top of a layer of commercial standards and protocols, which interfaces to commercial off-the-shelf automated test equipment. The LB and STA

structures are very similar because they are both solutions to the same problem. The purpose of a test is independent of which ATE is being used to exercise that functionality, so test information reuse can be increased by isolating the test from the specifics of the hardware setup. This is the purpose of the API layer of libraries in the STA system and the top level of services in the LB system.

Both systems also recognized the need to support commercial off-the-shelf hardware. This hardware was already in use (being controlled manually) and the inability to use already-available hardware would have been an unnecessary cost. Also, the ability to use off-the-shelf hardware increases the likelihood that the ATS and LB systems will be useful to other divisions within the company and to other companies altogether.

The final interesting similarity between the two systems is the ability for test designers to bypass the device-independent layer of the system and go directly to the device-dependent layer. There is the possibility that the test designer is more familiar with the ATE being used than the designers of the STA or LB systems, and wants to exercise some device-dependent functionality that wasn't included in the device-independent library level. By calling device-dependent library functions directly, the test designer can exercise ATE functionality that the STA developer may not even have known about. He can also help direct the expansion of the device-independent libraries in useful ways.

Chapter 3
Development environment

This chapter describes the environment in which the STA system was developed and the first product line it as applied to.

3.1 Work environment

The STA system was developed at Hughes Network Systems (HNS) in the Telephony Earth Station (TES) division. The system was designed and developed by a group of four people. Some additional input was provided from other employees. The development group consisted of three software developers (Braeton Taylor, Debapriya Sarkar, and me) and one manager (John Vespoli). At the beginning of my involvement, the requirements phase was ending and the design phase was about to begin. Approximately twenty-five man-months were scheduled for initial STA development [HNS2].

High-level system design was done in group meetings. These meetings discussed configuration management issues, separation of labor, structure of the system framework, structure of the library hierarchies, and the separation of generic and project-specific information. Subsystem design and coding was divided between the three developers. The subsystem designs were reviewed by the group in formal design reviews. The division of labor was as follows.

John Vespoli was responsible for the overall direction of the project. He authored the requirements specification for the system and participated in the high-level design discussions. In addition, he made the final decision in the event of any design conflicts. He was also responsible for obtaining the resources required for the project.

Braeton Taylor was responsible for the design and implementation of the generic GUI libraries and the STA framework. He was also responsible for the libraries and functions to exercise the command-line interface of SUT software. Braeton is a member of the Software Tech (SWT) group at HNS, which will be responsible for maintenance and control of the generic portions of the STA system.

I was responsible for the design and implementation of the generic hardware libraries and for promoting code reuse within the system. Ensuring the system allowed for the maximum reuse included the generation of standards and requirements for naming and documentation.

In accordance with the idea of separating the generic and project-specific information, the generic and project-specific labor was separated. Debapriya Sarkar was responsible for designing and implementing all TES-specific elements of the system. He was responsible for creating test plans, tests, sequences, and project-specific libraries that would execute the tests listed in section 1.1.1.

3.2 Testing environment

Figure 3.1 shows a picture of the development and execution environment used for the STA system. The STA system operates on a dedicated Unix workstation, which must not receive any other mouse or keyboard input while STA is running. The majority of STA is written in the Hewlett Packard Visual Engineering Environment (HP VEE), including the STA framework, which controls execution flow. The framework reads and writes all testing artifacts to an STA directory tree, which is controlled by the ClearCase configuration management system.

STA communicates with the hardware and software of the SUT through either a software test tool or external test hardware. Communication with SUT software is done using Mercury Interactive's XRunner package. XRunner allows the simulation of mouse and keyboard input from an operator, and is capable of reading text back from a SUT's GUI to determine test results. The GUI of the system under test is projected onto the display of the workstation STA is running on. Projecting the GUI onto the local STA host has the benefit that STA is independent of which operating system the SUT is running on, as long as its GUI can be projected onto a Unix workstation. STA communicates with external test equipment via serial ports or terminal servers by using the hardware library functions.

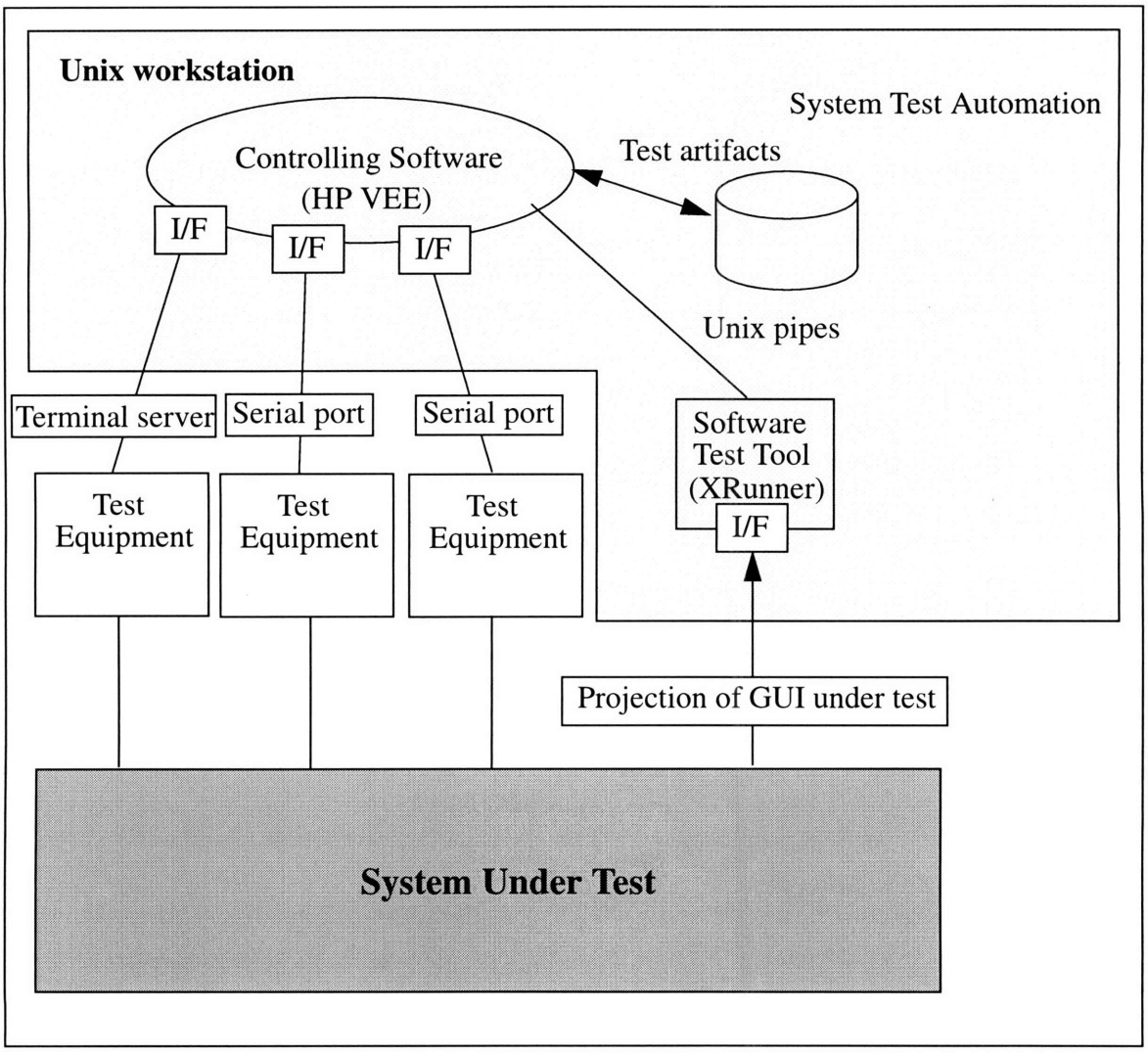

Figure 3.1 Test development and execution environment

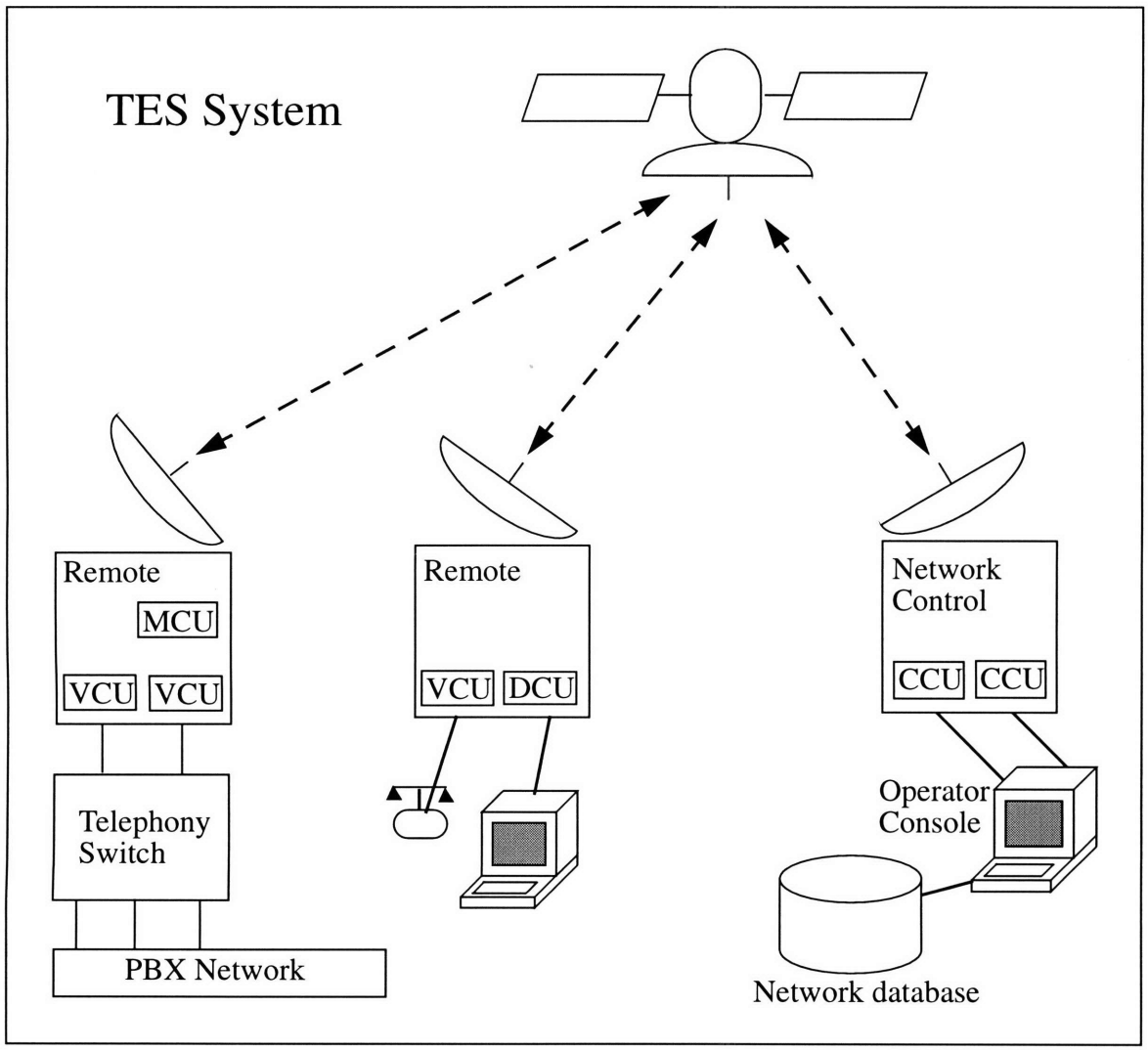

Figure 3.2 TES system layout

3.3 The TES system

Figure 3.2 shows a high-level picture of the Telephony Earth Station (TES) system. TES is a satellite telephony network. It is similar to other phone networks, except that it uses satellite links for network connectivity rather than the typical underground telephone lines. TES is capable of voice transmission using a number of telephony signalling protocols, and data transmission at various speeds.

A TES network consists of one or more network control sites and many remote sites, linked through one or more satellites. A remote site consists of a number of *channel units*

(CUs). The voice channel unit (VCU) connects to a telephone line at the remote site, which could go to a telephone handset, telephony switch, protocol converter, or other equipment. A single VCU can be assigned zero, one, or many telephone numbers. For example, a single VCU can correspond to an entire telephone exchange. Data channel units (DCUs) connect to remote data terminals or computers through serial lines. Monitor channel units (MCUs) allow the network control sites to monitor the operation of remote sites. Other types of channel units are available at the remote site, but they are not of great interest here.

The network control sites direct the operation of the telephone system. They allocate and deallocate bandwidth, assign frequencies for telephone connections, set up and tear down data connections, monitor remote site status, and record network statistics. All communication between remote sites must be set up and registered with a network control site through control channel units (CCUs). The CCUs also provide the ability to update the configuration and software of remote CUs from the network control site. The network is controlled through the network control site's operator interface.

Network control is based on a network database. The network database contains all information required to run, monitor, and debug the network. This includes tables of satellite link parameters, remote site and channel unit definitions, bandwidth allocations, configuration information, billing information, network statistics, call records, etc. Without a corresponding entry in the database, any physical element does not exist in the network.

Chapter 4
System Design

This chapter describes the design and implementation of the STA system.

4.1 Design overview

Figure 4.1 gives an overview of the STA system [HNS3]. The system has both generic and project-specific parts, which are differentiated in the figure by shading. This section describes each part of the system in minimal detail to give context for the following sections.

The operator interface sits on top of sequences, tests, and function libraries, and handles all communication between STA and the tester. It executes the sequences and tests developed for the system under test. The operator interface allows the tester to enable, disable, or configure tests and sequences at run-time. It is also responsible for presenting the results of the test run to the tester.

The framework is the driving force behind the operator interface. All operator interface functionality is provided by the framework. The framework is also responsible for loading and deleting sequences, tests, and hardware libraries to and from memory.

Sequences and tests are themselves function libraries which are loaded and deleted to and from memory. There are a number of external files and global variables required by tests and sequences.

Below the sequences and tests are function libraries, both project-specific and generic. They are loaded into memory for tests and sequences that need them and removed from memory when they are not needed. There are many different types of libraries: high and

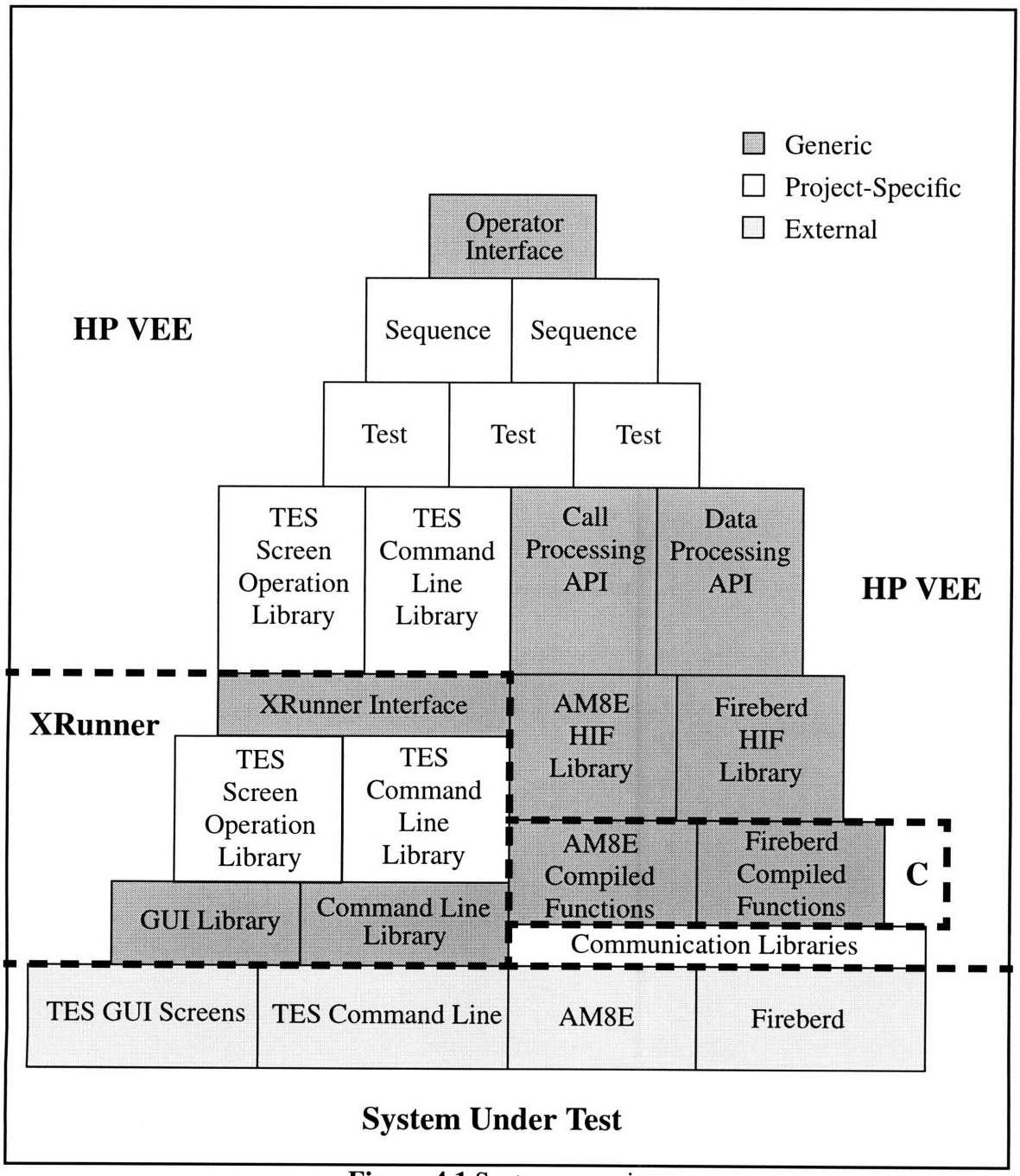

Figure 4.1 System overview

low-level, generic and project-specific, APIs, HIFs, compiled libraries, XRunner libraries, HP VEE / XRunner interface libraries, and others. Each type of library serves a different purpose.

The system uses many global variables and text files. The global variables are used to store hardware configuration, which tests and sequences are enabled and disabled, and the properties of the GUI of the system being tested. The text files are used to store default parameter values, results, and other information.

The STA system uses two processes on the host workstation. The controlling process is the HP VEE process running the STA operator interface and framework. In addition there is an XRunner process which handles communication with the SUT's GUI. The XRunner process is fed commands from the HP VEE process through Unix named pipes.

The directory structure of each instantiation of the STA system is project-specific and can be customized. In general, generic and project-specific libraries will be in different sub-trees, as will HP VEE libraries and XRunner libraries. The tests and sequences developed for the SUT are also part of the directory tree.

4.2 Operator interface

This section describes the interface between STA and the test operator [HNS4]. The framework that implements all the functionality behind this interface is described in section 4.3, and the role of tests and sequences in this process is described in section 4.4. The STA system can be run in either interactive mode or batch mode, depending on whether an operator is present for the beginning of test execution.

4.2.1 Interactive mode

Interactive mode is used when there is an operator present for the beginning of the test run. In this mode, the operator is presented with a series of screens allowing him to configure the test run and begin its execution. Before test execution, the operator must start the system under test and project its GUI onto the workstation running STA. He must also start an XRunner process running the XRunner script 'control' (explained in section 4.5.3). The first screen presented to the operator is a welcome screen presented while the rest of STA is loaded. The operator interface screens and possible transitions between screens are shown in Figure 4.2. The internal libraries used to generate operator interface screens are described in section 4.3.

Set-up screen

The STA system needs two file names to begin its set-up: a sequence file and a configuration file. Upon execution, STA looks for a two-line file named 'STA.setup' containing these file names. If interactive mode was specified when STA was run, or if the 'STA.setup' file was not found, the operator will be presented with a series of dialog boxes allowing him to chose which sequence and configuration files to use. Once this has been done, the operator is presented with the system configuration screen.

Main screen

The main screen of STA allows the operator to perform a number of actions, either directly or through sub-screens. The main screen contains summary information including: the name of each sequence in the test run, the number of tests selected from each sequence, the number of times the test run will be repeated, and the conditions on which STA execution will pause.

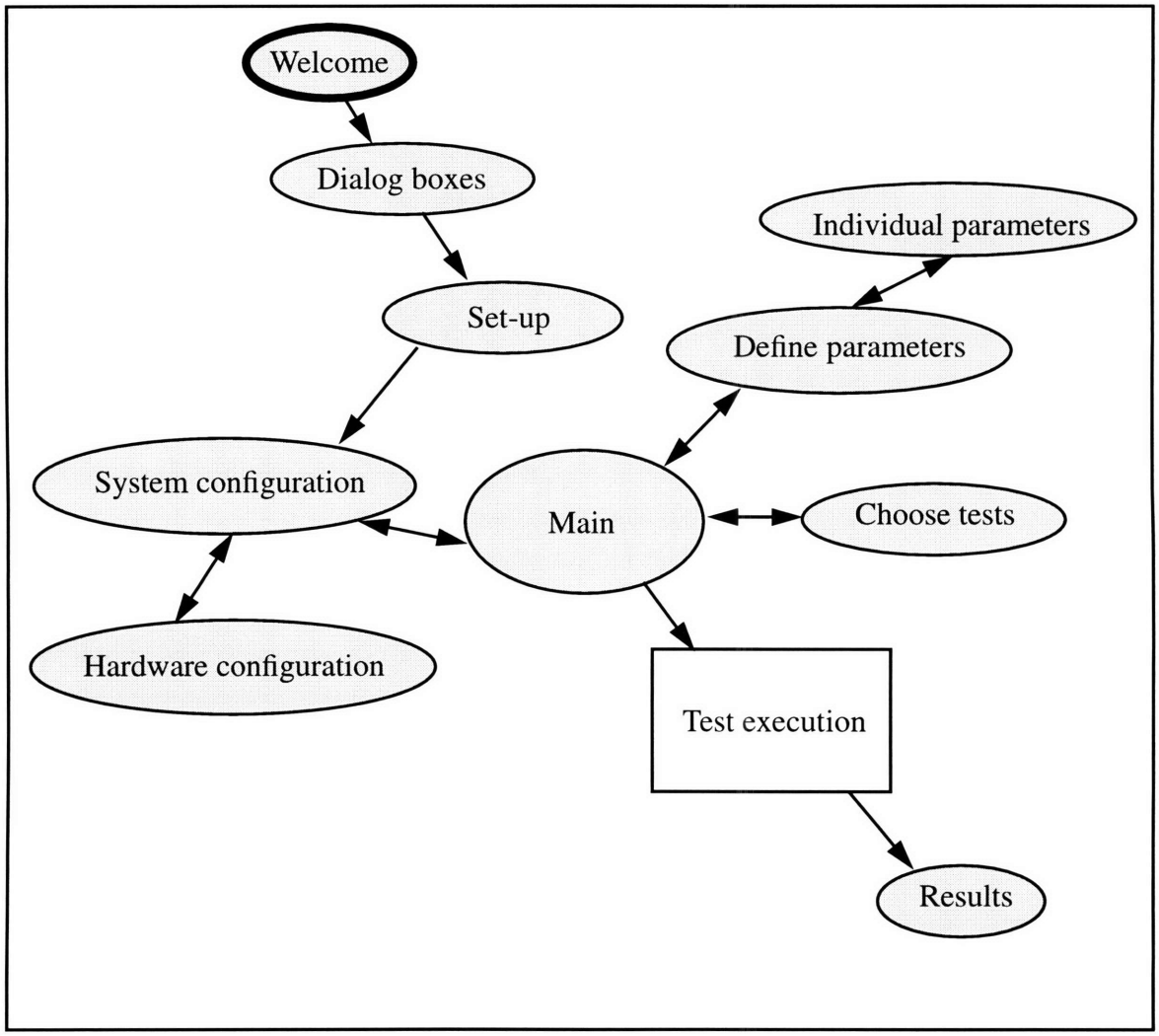

Figure 4.2 STA operator interface structure

From the main screen, the operator can:

1. **Select and deselect which tests to run**. The main screen has buttons to enable or disable entire sequences. In addition, the operator can go to the *Choose tests* screen for each sequence to enable or disable individual tests.

2. **Define test parameters**. The operator can access the sequence-specific *Define parameter* screen for each sequence. From there he can access *Individual parameters* screens to set test parameters for each test in the sequence.

3. **Change the repeat count setting**. The entire test run can be repeated by setting the repeat count. The repeat count is typically set to one.

4. **Change the pause setting**. The pause setting determines the conditions on which test execution will pause for operator confirmation. This can be set to: 'don't pause', 'pause between sequences', 'pause between tests', or 'pause on test failure'.

5. **Show which tests are selected to be run**. The operator can generate a listing (by sequence) of all the enabled tests.

6. **Configure the STA system**. The *System configuration* screen can be accessed to change system parameters.

7. **Begin the test run**. This begins the execution of the selected tests, and eventually leads the operator to the results screen.

The *Main* screen is created by a call to an internal library function. Each of these libraries contains a single function that creates and displays a *Main* screen with a different number of sequences. The framework handles loading the correct library and calling its function with the appropriate sequence names.

Choose tests screens

Each sequence has its own *Choose tests* screen. This screen gives a name and a check box for each test in the sequence. The operator selects which tests to run in the sequence via the check boxes. The *Choose tests* screens are created by functions in internal libraries, which are called with the appropriate inputs by the framework.

Define parameters screens

Each sequence has its own *Define parameters* screen. This screen gives a name and button for each test in the sequence. Clicking on a test's button takes the operator to the *Individual test parameter* screen for that test. The *Define parameters* screens are created by functions in internal libraries, which are called by the framework with the appropriate inputs.

Individual test parameter screens

There is one *Individual test parameter* screen for each test in each sequence. This screen gives the test name, and text entry boxes for each parameter, allowing the operator to set their values. Default parameters are loaded from files. The operator also has the

options of loading parameters from a file and saving the current parameters to a file. These screens are created by the test itself, and are described in section 4.4.2.

System configuration screen

The *System configuration* screen allows the operator to change parameters for the STA system. It allows the operator to set the path to search for default test parameter information, the directory to store test results in, and the hardware configuration file name. It also allows the operator to select the name of the file where default parameters are stored for each test defined in the test run. The *System configuration* screen has buttons to display the *Hardware configuration* screen, load system configuration from a start-up file, set all test parameters to their defaults, and save the system configuration to a file.

Hardware configuration screen

The *Hardware configuration* screen allows the operator to define the external test hardware to be used in the test run. It lists all supported external hardware types. For each piece of external hardware, it allows the operator to define the logical identifier of the equipment, the type of hardware it corresponds to, the method of communication used to control the hardware, and hardware-type-dependent data fields. Hardware configuration will be explained in more detail in section 4.3.

Results screen

The Results screen displays test results at the end of a test run. The display includes the number of test run, number of tests passed and failed, whether the STA system completed its run normally or abnormally, and a "test status line" for each test (described in section 4.5.3).

4.2.2 Batch mode

Because the STA system was designed to run in the absence of any operator, it can also be run in "batch mode." To do this, a start-up file (or files) is created, as before, as well as a Unix script file. The Unix script simply copies a start-up file into STA.startup and executes HP VEE and the STA main program. STA is capable of loading all parameters and executing the test run given only this start-up file. Using a Unix script allows the concatenation of multiple test runs by defining multiple start-up files and copying each into STA.startup in turn.

4.3 Framework

The framework is the engine behind the operator interface. In addition, it is responsible for the control of the high-level execution flow through the system [HNS5]. The majority of the framework is contained in the Utilities, Choices_Panels, Define_Panels, and Main_Panels libraries. The Utilities library holds functions that are used by the STA system to configure the system, test run, and hardware, modify global variables, handle external files, execute tests and sequences, and load and delete libraries. The Choices_Panels, Define_Panels, and Main_Panels libraries contain functions to construct and display operator interface screens with sequence and test names as labels. The internal libraries allow the framework to be stored as a group of functions, which provides modularity and eliminates duplication of code.

4.3.1 System configuration

The configuration of the STA system is handled by the framework, both at the beginning of execution and by operator request. System configuration information is stored in global variables so it is available to the entire framework. The information includes the

directory path to locate default test parameters in, the default parameter file for each test, the directory to place test results in, and the hardware configuration.

4.3.2 Controlling sequences, tests, and execution flow

The main responsibility of the framework is to control the running of sequences and tests during each run. When it is started, the framework loads the sequence file specified on the STA *Set-up* screen. This file contains a single function which stores sequence and test names in global variables. The sequence and test names are used to create labels on the operator interface screens and determine the names of files containing functions used to set up, execute, and clean up each sequence and test. Using the information in the global variables, the framework generates and displays the STA *Main* screen.

Once the test and sequence information is loaded into the STA system, the operator is able to select or deselect which tests will be included in this test run. This is done by modifying another global variable which contains flags determining whether each test will be run or not. This variable is modified in the Choose tests screen.

The operator is also able to change test parameter values. When the operator goes through the *Define parameters* screen into the *Individual test parameters* screens, the framework uses the test information to locate the file in which the test resides. The framework then calls the Define_<Test> function (explained in section 4.4.2) defined in that file which allows the operator to modify parameter values. All test parameter values are stored in global variables.

When the operator clicks on the "RUN" button on the *Main* screen, the framework calls functions to perform the test run. This involves loading the sequences and tests into memory and calling one or more functions defined in them. The test developer has a num-

ber of optional functions that he can define for any test or sequence. The framework will look for these optional functions and call them (if they exist) at the appropriate times. In particular, the test developer can define functions that will automatically be run before a given sequence, after a given sequence, before a given test, or after a given test. These optional functions allow STA to dynamically alter the testing sequence depending on the results of previous tests. For information on how this is done, see the description of tests and sequences in section 4.4.

Each test that is run returns its results to the framework upon completion. The framework records results and keeps statistics including: where in the test run it is, which pass of the complete test run it is on, number of tests passed, and number of tests failed. Depending on the value of the pause setting, the framework will wait for operator confirmation on different events in the test run. Most of the libraries required to run each test or sequence are loaded and deleted in the functions called before and after sequences (see section 4.4). Because of their nature, however, some hardware libraries are loaded by the framework itself.

4.3.3 Hardware configuration

The default hardware configuration is loaded from the file name specified in the STA start-up file. Because the hardware configuration doesn't change during a test run, the framework loads the appropriate lower-level hardware libraries depending on the hardware configuration. This lessens the burdens on the test developer.

Hardware configuration is concerned with two types of information, the supported hardware types, and the hardware present in the current configuration. Because of this,

hardware configuration is centered around two global variables, GL_HW_Types and GL_HW_Config.

GL_HW_Types identifies the types of external test hardware supported by STA and contains the file names of the libraries associated with each hardware type. GL_HW_Types stores 4 fields for each supported type: the device type name (AM8e or Fireberd in the TES STA system), the location of the Hardware Interface (HIF) library for that type of equipment, and the location of the two files needed to provide a compiled function library (CLIB) for use by that HIF. If compiled libraries are not needed for that hardware type, the last two fields can be left blank. The information in GL_HW_Types is presented to the tester as read-only information. It is assumed that if the tester knows enough to properly add another hardware type to the STA system, then he will be able to edit the default configuration file by hand to list the new hardware type in GL_HW_Types.

GL_HW_Config stores information on the external test hardware physically present in the test bed. GL_HW_Config can be modified by the test operator because it is likely to change and because modifying it requires less detailed knowledge about the STA system. Currently, STA can control up to 16 pieces of external hardware. For each piece of external hardware defined in GL_HW_Config, the following information is stored:

1. **Hardware ID**. This is the logical name of the piece of hardware, and it is the label by which all tests and libraries refer to the hardware.

2. **Hardware Type**. This is the type of the piece of hardware, as defined in GL_HW_Types.

3. **Active?**. This is a flag for whether the piece of hardware is currently active or not.

4. **Communication information**. This is the means and address used to communicate with the hardware ('terminal server' or 'serial', IP address and port, serial port number, baud rate, etc.).

5. **Hardware-specific information**. The meaning of this information depends on the type of the hardware.

Associating each piece of hardware with a logical hardware ID (HWID) gives tests a compact way of specifying which piece of hardware they are referring to. It also allows tests to be independent of which type of hardware they are operating. Given a HWID, any test or library function can retrieve all of its configuration information by use of a function in the Utilities library. None of the tests need to be changed if a new type of hardware is supported by the system. Once the new HIF library is added to the system and the new hardware type is identified in GL_HW_Types, all that needs to be done is to change the information in GL_HW_Config and the tests will be able to run on the new hardware.

The final responsibility of the framework in hardware configuration is to load some low-level hardware libraries. Unlike most libraries, hardware libraries are loaded at the beginning of the execution run and stay present in memory for the duration of the run. As stated above, this is done because the test bed configuration should not change during a run of the STA system. The framework identifies the types of all the hardware present in the test bed from GL_HW_Config, and loads the HIF (and possibly CLIB) specified in GL_HW_Types for each type of hardware present. These libraries are deleted from memory by the framework when it terminates.

In the same way as it determines the HIFs and CLIBs required by the current test bed configuration, the framework determines which communication libraries are required. The communication libraries handle the physical transfer of commands to external hardware. Depending on the communication information stored in GL_HW_Config, not all of these may be needed. In addition, loading some of these libraries when they are not needed will

cause run-time errors (see section 4.5.4). It always loads the Ports communication library because it is always used in communicating with external hardware.

4.4 Sequences and tests

4.4.1 Sequences

The definition of sequences is done in the sequence file specified in the STA start-up file. At a minimum, this sequence file must contain a function which creates the global variables that store the list of sequences and test names to be recognized by STA. This is all that must be done to define sequences of tests in STA.

The sequence file is responsible for determining when most libraries are loaded and unloaded. Because tests are treated as libraries, the sequence file is also responsible for determining when tests are loaded and unloaded. There are three options for when to load and unload libraries and tests [HNS5].

Load everything at the beginning of STA execution

The developer of the test run has the option of loading all libraries and / or tests at the beginning of STA execution and keeping them in memory until the end of STA execution. STA looks for initialization and termination functions in the sequence file. The Initialize function is called at the beginning of a test run and the Terminate function is called at the end of test execution. The developer can load libraries and tests in the Initialize function and delete them in the Terminate function, causing them to remain in memory throughout the entire test run.

Load everything sequence-by-sequence

Rather than loading everything into memory in one fell swoop, the developer can design his test run to load libraries and tests sequence-by-sequence. STA looks for

optional functions (whose names are constrained by STA) in the sequence file that are run before and after a specific sequence. These functions have a similar purpose as Initialize and Terminate, except that they are executed around a particular sequence rather than around the entire test run. Tests and libraries loaded and deleted in these functions remain in memory only during the execution of one sequence.

Load everything test-by-test

The method used to load libraries test-by-test is analogous to the one used to load them sequence-by-sequence. STA looks for optional functions (whose names are constrained by STA) in the sequence file that are run before and after a specific test. These functions have a similar purpose as Initialize and Terminate, except that they are executed around a particular test rather than around the entire test run. Libraries loaded and deleted in these functions remain in memory only during the execution of one test. These functions are used to load and delete libraries for individual tests, but are not used to load and delete tests. The default action is for the STA system to load tests right before they are run and delete them once they are completed, so no extra steps must be taken for this to happen.

Comparison of the options

Each of the three options for loading and deleting tests and libraries is appropriate for a different situation. The final choice is a trade-off between optimizing for memory, speed, and coding time.

Loading everything at once is a good choice if you have a small number of sequences and tests defined. In this situation, the tests and libraries are unlikely to exceed the memory available, so there is nothing lost by loading everything at once. In addition, loading everything at once takes the least coding time and maximizes the speed of the test run.

This can also be useful when a single test is run in different sequences under different initial conditions because the test will not be deleted and reloaded.

Loading libraries sequence-by-sequence is useful in larger projects if tests within a sequence are likely to use the same libraries but tests in different sequences are unlikely to use the same libraries. While some time may be lost reloading libraries used by more than one sequence, execution time within a sequence is still optimized. Loading sequence-by-sequence is also a good compromise if there are enough sequences and tests that loading the entire test run would exceed the available memory or severely degrade performance, but each sequence is short enough that loading one sequence at a time will not affect performance.

Loading libraries test-by-test is useful in a few situations. The first is when the size of sequences and complexity of tests or libraries makes it unattractive to load all the libraries for an entire sequence at once. In this case, the time lost by deleting and reloading libraries may not be as much as the time gained by better performance. It is also useful when only a few tests use external libraries or the tests in a sequence each use different libraries.

It is allowable to choose different options for loading tests and loading libraries, and it is also possible to combine the methods for loading libraries. Doing this can provide the best memory / performance trade-off for a particular test run, but greater care must be taken to ensure that libraries are present when they are needed and aren't left in memory at the end of execution.

4.4.2 Tests

Each test is contained in one HP VEE file. One function is required for each test, and four more are required if the test has parameters that can be set at run-time [HNS5]. For

each test, <Test> is the name of the test as found in the global variables, and the test functions are stored in the file <Test>.test. The required function for every test is Run_<Test>, and the additional functions for configureable tests are Defaults_<Test>, Define_<Test>, Load_<Test>, and Save_<Test>.

The Run_<Test> function performs the test and returns the results to the framework. It is called by the framework at the appropriate time if the test was enabled by the operator. The test result is a character string in the following format:

```
("PASS"|"FAIL") <Status_Code> [<Additional_Textual_Information>]
```

The format of this string is described in more detail in section 4.5.3. The Run_<Test> function can use calls to libraries to control external hardware, operate XRunner functions, or perform any other subroutines.

The Defaults_<Test> function creates and assigns default values to any global variables that are needed by the test, including all parameters that the operator can change in the *Individual test parameters* screen for this test. This function is called by during STA initialization, and whenever Define_<Test> or Load_<Test> return an error.

The Define_<Test> function creates and presents the *Individual test parameters* screen for the test, allows the operator to change the value of parameters for the test, and performs error checking on the operator's input. The default values shown to the operator when the screen appears are received from Load_<Test> (or Defaults_<Test> when Load_<Test> returns an error).

The Load_<Test> and Save_<Test> functions exchange parameter values between files and global variables. The Load_<Test> function loads parameters for this test from a file and saves their values in global variables. The Save_<Test> function does the reverse.

4.4.3 Dynamic alteration of the test sequence

In some cases, dynamically enabling or disabling tests during execution can be very useful. This is true when some tests can be invalidated depending on the results of previous tests, or when you want to run diagnostic tests only when an error is detected. Dynamic sequence alteration is done by adding optional functions to the sequence file.

To perform dynamic alteration in the middle of a testing run, the STA framework looks for more optional functions. They are added to the sequence file, and are almost exactly like the optional functions used to load and delete libraries and tests. Thus the dynamic alteration functions can be run before or after any specific sequence or test. The purpose of the dynamic alteration functions and the library loading functions are unrelated, however, so the library loading functions should not modify the testing sequence, and the dynamic alteration functions should not load or delete libraries or tests.

The dynamic alteration functions alter the sequence by using functions from the framework's Utilities library. These utility functions allow the dynamic alteration functions to observe the results of a test, enable a test, or disable a test.

4.5 Libraries

4.5.1 Overall library structure

Figure 4.3 shows the structure and types of libraries in the STA system [HNS5]. The library structure is central to the design of the system. Much of STA's flexibility and reusability is a result of this structure. This structure allows incremental expansion of STA's functionality in a well-organized manner. Because many libraries are functionally independent, the addition or modification of a library is unlikely to affect many other parts of

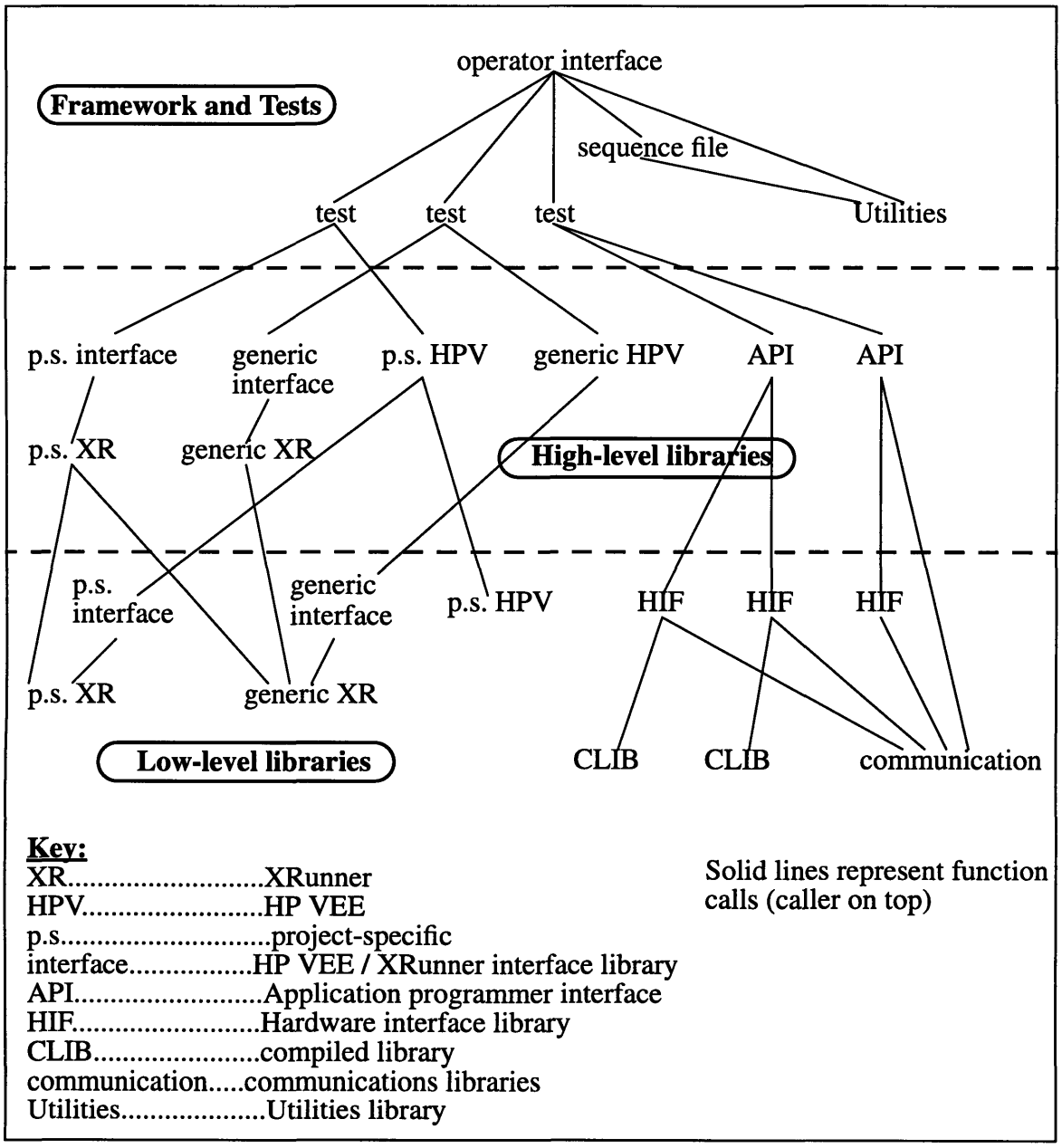

Figure 4.3 Overall STA library structure

the system. Figure 4.4 shows the generic libraries included in the initial STA implementation [HNS6].

Libraries are implemented as either HP VEE *User Functions* or XRunner *functions*. These language constructs provide (in their respective languages) a number of benefits over other choices. HP VEE *User Functions* provide modularity, and are analogous to

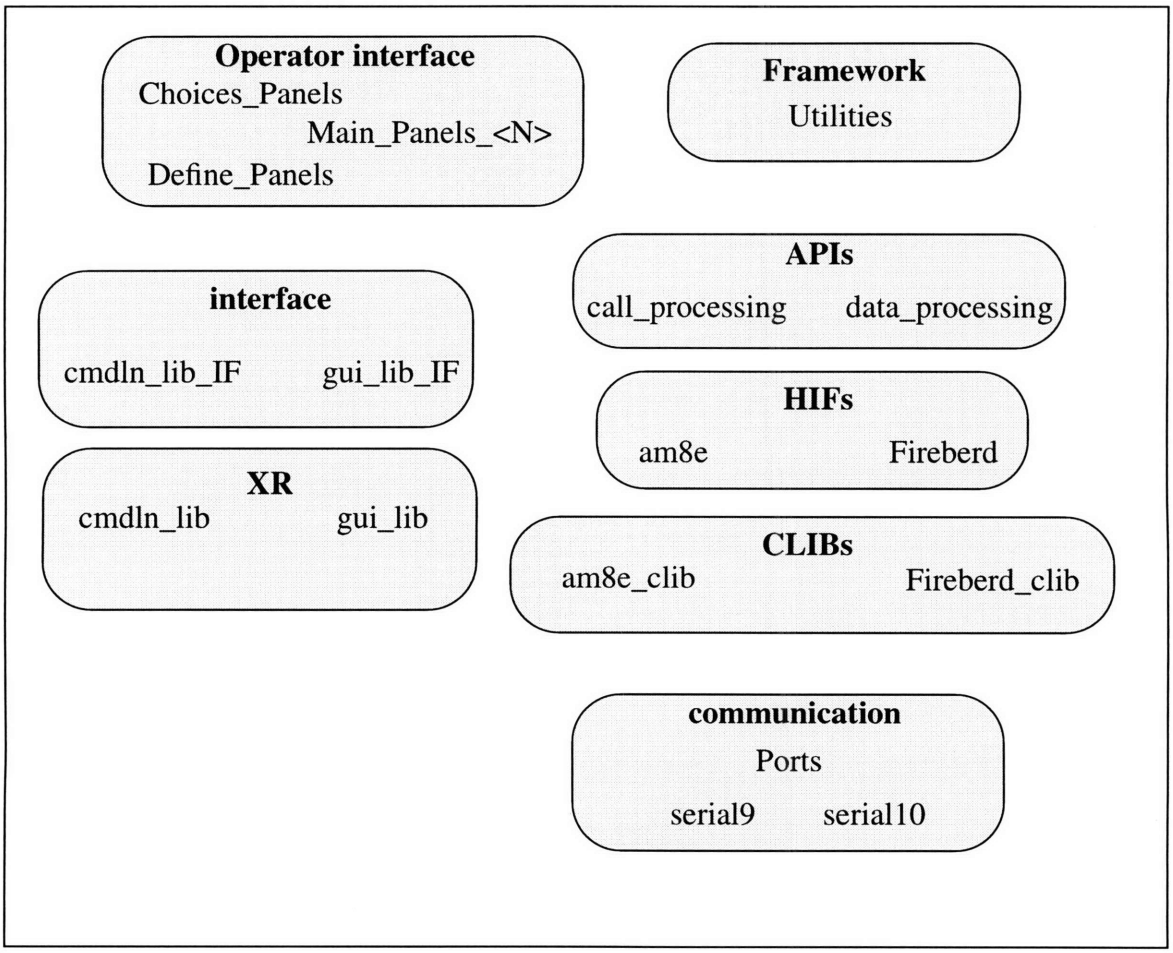

Figure 4.4 Existing generic libraries

functions or procedures in textual languages. The other modularity construct in HP VEE is the *User Object*. *User Objects* were not chosen because each instantiation of a *User Object* is a separate copy of HP VEE "code". Thus a change to a *User Object* would need to be repeated for every occurrence of that object within the system, which is clearly unacceptable. *User Functions* increase the system's maintainability and save memory by only keeping one copy of common code in the entire system. Also, *User Function* libraries can be dynamically loaded into memory while *User Objects* can not. Dynamic library loading also conserves memory by allowing code to be loaded in memory only while it is being executed. XRunner *functions* are analogous to HP VEE *User Functions*. They were cho-

sen over XRunner *scripts* (which are similar to HP VEE *User Objects*) for the same reasons.

The libraries in the STA system can be partially differentiated by the "level" at which they provide functionality. The highest level of functionality and control flow is provided by the framework, sequence file, and tests. Most of the framework's functionality is implemented in the Utilities, Choices_Panels, Define_Panels, and Main Panels libraries. The sequence file handles loading and unloading of lower-level libraries and modification of the testing order, and the tests define the process to use in each separate test on the SUT. High-level and low-level libraries are separated by the scope of the actions they perform. Low-level libraries perform simple tasks, and high-level libraries are built on top of low-level libraries to perform more complicated tasks.

STA libraries can also be classified as either generic or project-specific. Project-specific libraries are tied to the particular SUT for which they were designed. They contain functions that are likely to be reused within a collection of test for a particular SUT, but are unlikely to be useful on a different SUT. Generic libraries contain functions that may be useful across a number of projects and SUTs. Because generic libraries must be independent of the SUT, they can not contain calls to any project-specific library functions. It is perfectly acceptable, however, to have project-specific functions that call both project-specific and generic library functions.

The final means of classifying libraries is by the type of functionality they provide. Each different type of library in the system will be discussed below.

4.5.2 HP VEE libraries

These libraries contain HP VEE User Functions that may be useful to more than one test, either within a single project or across multiple projects. These "standard" HP VEE libraries are better defined by what they do not do than by what they can do. They do not provide a direct interface to either external hardware or to XRunner functions, otherwise they would be in a different class of libraries (see sections 4.5.3 and 4.5.4). Aside from that, there are no other constraints on what they can do. Often these libraries will provide more complicated functions that are built on top of hardware interface and XRunner interface functions. Possibilities include: complex configuration of the SUT, configuring two call generators and placing a call between them, sending test data and checking that the bit-error-rate is below some specified threshold, and logging on to the SUT with a specified user name and password. The main purpose of these libraries is to provide modularity within tests and increase code reuse between tests.

4.5.3 XRunner and HP VEE / XRunner interface libraries

XRunner libraries contain XRunner functions that can be used across different tests, and possibly across different projects [HNS5]. In order to simplify the role of the test developer, HP VEE / XRunner interface libraries were included in the system. Each HP VEE / XRunner interface library mirrors an XRunner library, and contains a *mirror function* for each XRunner function in that XRunner library. The interface library functions just call the appropriate XRunner functions and return the results of the XRunner function. They format the inputs and outputs of the XRunner functions and deliver them over the named pipes used to communicate with XRunner. Interface libraries allow the test developer to call XRunner library functions in his tests in exactly the same way he calls other

library functions, eliminating the need for him to know about XRunner or communication through named pipes.

Related global variables

XRunner libraries operate either the command line interfaces or the graphical interfaces of the SUT. For XRunner to be able to operate on the SUT's GUI screens and to be able to gracefully accommodate changes to those screens, they are described in the STA system in a number of global variables [HNS6]. The global variables relate the menus, buttons, and text boxes on the SUT screens to their location in pixels. Thus the developer does not have to worry about the pixel location of each widget. Also, changes to a GUI screen affect only the global variables describing that screen, and do not require changes to any library functions. The global variables store information on the GUI screens that is sufficient to identify them by logical name, refer to their buttons, labels, and menus by name, and read text back from the screens to return to HP VEE.

The "control" script

The "control" script is the primary interface to HP VEE within XRunner [HNS5]. It is the script that is run when the XRunner process is started. It initializes the XRunner side of STA and then handles the calls to all XRunner library functions.

The control script first creates the Unix pipes used for communication with HP VEE and then initializes the global variables that are needed for communication between the two processes. It then calls a project-specific initialization file which contains code to initialize more global variables and load any XRunner function libraries that will be needed.

The control script dispatches requests from HP VEE by running the correct XRunner function with the specified parameter values. Any parameters required for the XRunner

functions being called are passed from HP VEE to the function through the control script. Commands and their parameters are passed between XRunner and HP VEE as character strings. Calls to XRunner functions from HP VEE have three parts: the command category, the command name, and parameters (if required). The purpose of the command category is to modularize the control script. The control script branches to subroutines depending on the command category. Each subroutine handles function calls within one category, resulting in a simpler control script. The command name tells the control script which XRunner function it should call, and the parameters are passed to that particular function.

The response from XRunner to a request from HP VEE is known as the *test status line*. This is a single line of text, and follows the format:

```
("PASS"|"FAIL") <Status_Code> [<Additional_Textual_Information>]
```

The first portion is an indication of whether the test passed or failed. The <Status_Code> is a three-digit integer giving information about why the test passed or failed, and the <Additional_Textual_Information> is an optional text string to be displayed on the results screen and stored in the test results file. It is known as the test status line for two reasons. The format of the return from an XRunner function is exactly the format used to collect test results, and often the results of a test are copied directly from the return value of an XRunner function used to perform the test.

4.5.4 Hardware libraries

Libraries that deal with interfacing to hardware fall into four categories: Application programmer interface (API) libraries, Hardware interface (HIF) libraries, compiled function (CLIB) libraries, and communication libraries [HNS5]. API libraries provide the

developer with a common interface to different hardware types. HIF libraries contain functions that are called by the API library functions (or directly by test developers in some cases) that handle hardware-type-specific issues. CLIB libraries contain compiled routines written in a procedural language (like Pascal, PLM, or C) that are usually called by HIF functions. Currently, compiled function libraries are only used by HIF libraries, which is why they are included in the hardware library structure. For a more detailed description of the role and structure of compiled libraries within the STA system, see section 4.5.5. Communications libraries encapsulate the details of sending text strings over serial ports and through terminal servers. The structure of the existing hardware-related libraries as they relate to tests is shown in Figure 4.5. Hardware libraries can be added in any of the four categories. The figure shows only existing libraries to give a more concrete explanation of the hardware library structure. Because these functions are intended for widespread reuse, documentation is even more important than usual. Documentation standards are described in section 4.6.

API libraries

The purpose of the API libraries is to make the developer's tests as independent of hardware changes as possible. Tests should not need to be rewritten when one type of hardware replaces another with the same functionality (two different types of data generators, for example). In addition to this, the API libraries must provide the freedom and functionality for the test developer to create meaningful tests. They should give access to all the functionality of the test hardware that the developer is likely to use, and they should do so while hiding machine-specific issues.

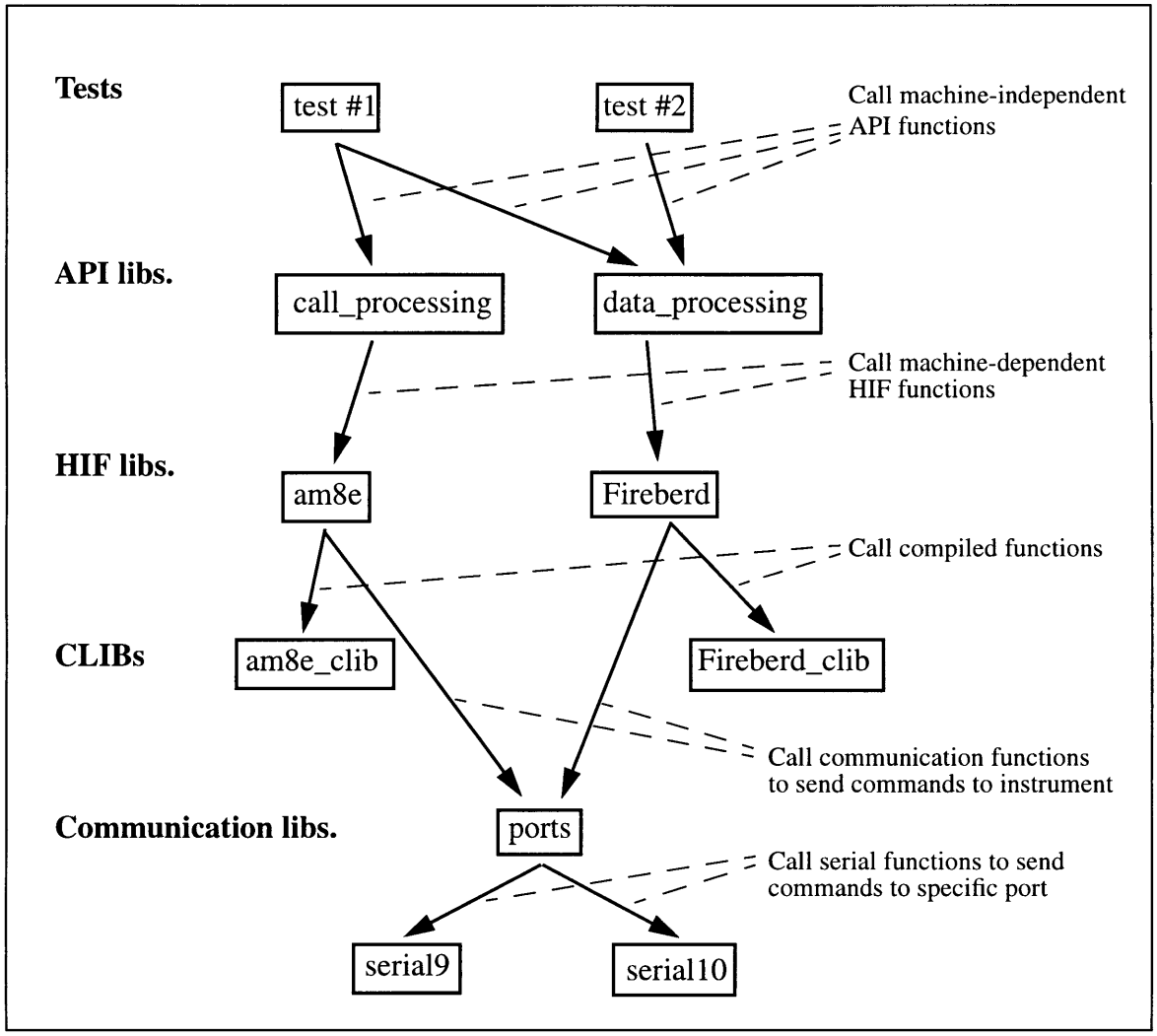

Figure 4.5 Hardware library structure

API libraries are structured around the types of operations they perform. The call_processing API library contains a group of functions all related to controlling telephone call generators. The data_processing library, on the other hand, contains only functions related to controlling data generators. A piece of test hardware that simulates voice traffic is more likely to share functionality with other hardware that simulates voice traffic than it is to share functionality with hardware that simulates data traffic. So grouping API libraries around functionality (call_processing, data_processing, etc.) also structures them around the test hardware's functionality. Obviously, API functions can not be completely

independent of hardware (a function that sets up for R2 telephony signalling cannot be executed without hardware capable of R2 signalling) but as long as hardware with the proper functionality is present, the API functions operate the same no matter which device type is being used.

To allow tests to be independent of hardware changes, API functions identify hardware devices via logical hardware IDs (HWIDs). The mapping between a HWID and a physical piece of test equipment is defined in the *Hardware configuration* screen of the operator interface. The API functions access all hardware information through the logical ID number and global hardware configuration variables a function from the Utilities library. This function takes a HWID as input and returns all the configuration information available on that piece of equipment. Tests calling API functions never need to know hardware details. A call to an API function might be "seize_line(HWID_1)" or "init_data_generator(HWID2, 'burst', 'random')". The API functions are then responsible for choosing and calling appropriate hardware-specific HIF functions.

Because HIF functions are machine-specific, API calls to these functions will pass machine-specific parameters. Thus the transition from machine-independent to machine-dependent function calls occurs within the API library functions. Typically, API functions will take the form of a large 'case' construct, with a different branch for each supported hardware type. Each branch will call functions from a different HIF library with machine-specific parameters.

In addition to taking hardware-independent inputs (in the form of HWIDs), API functions must return hardware-independent outputs. Thus API functions are also responsible for reformatting HIF function output to be machine-independent. It is important to

remember that API libraries form an interface. Thus they are not responsible for making high-level decisions based on hardware output, only for providing that output to the calling function in a hardware-independent format.

HIF libraries

Hardware interface (HIF) libraries handle all steps specific to a type of hardware. They are responsible for checking the format of their inputs before they are executed on the hardware. Only valid commands are actually sent to the hardware. Sending invalid commands to the hardware can result in the hardware 'hanging', or it can cause input and output buffers to get out of synchronization. Either of these situations could ruin an entire overnight test run, so HIF functions ensure that all hardware commands are valid before sending them over the control ports.

HIF functions may also be responsible for exercising functionality unique to that hardware type, or for performing more complicated machine-specific actions. These could be things like clearing all the equipment's buffers and registers, or downloading files to the equipment. Each hardware type should have its own HIF library.

With the exception of the hardware type, HIF functions should be independent of the same things as API functions. Specifically, they should be independent of the mode of communication used to control the hardware. Because they are hardware-dependent functions, however, they do not need to reformat any of the output from the test hardware.

Compiled libraries

Compiled libraries are discussed in more detail in section 4.5.5. This section discusses their use in the hardware library structure.

Compiled libraries are used in the hardware library structure primarily to check the range and syntax of the commands passed from API functions to HIF functions. While this may not seem necessary (because API functions should be tested and debugged), checking range and syntax is needed when test developers call HIF functions directly. The cost of sending invalid commands to hardware during a long test run is high enough to justify the time and effort spent in assuring that only valid commands are sent to hardware.

Communication libraries

The communication libraries serve two purposes: they encapsulate the details of communicating with equipment over different types of ports, and they work around errors that HP VEE would otherwise generate depending on the configuration of the STA host.

Communicating over serial ports and terminal server lines (the two forms of communication currently supported) require different procedures in HP VEE. Incorporating the details of port communication into HIF library functions would make them overly complex, require multiple copies of the same code, blur the purpose of the HIF libraries, and reduce the ability to support new methods of communication.

Also, HP VEE can generate run-time errors if the serial ports of the host are not configured properly. If an HP VEE program contains an I/O object configured for a serial port (serial9, for example) and that port is not configured on the local host for use by HP VEE, then HP VEE will generate run-time errors regardless of whether execution flow ever reaches that I/O object. Serial port 9 is often configured to be a console port for the host, so this type of error is common. If the I/O object is in a library function, on the other hand, a call to that library function can be in the HP VEE program without generating an error as long as the library is never loaded into memory and execution flow never reaches the func-

tion call. By putting all serial I/O objects in the serial9 and serial10 libraries, the STA system has the ability to use different serial ports on different hosts without generating errors and without writing different I/O routines for each host. Finally, the STA system loads serial9 and/or serial10 during hardware configuration, so an operator can be assured that none of these errors will occur during a test run if they did not occur during hardware configuration.

4.5.5 Compiled libraries

Compiled functions are used when they provide a substantial improvement in either performance or library structure. Because HP VEE is an iconic language, computationally-intensive functions like parsing command strings and checking their validity results in very complicated and obscure HP VEE functions. In general, implementation of range and syntax checking is faster, easier, and clearer when done in a traditional functional language.

Compiled function libraries (CLIBs) let the developer include code written in other languages in the STA system. This lets him use functional languages when they will reduce execution time or decrease the complexity of other HP VEE functions. Using compiled functions, however, can increase the complexity of the system if it is not done properly, so they are only used when they provide a noticeable increase in performance or improvement in system design. To use a CLIB in HP VEE the developer must take two extra steps not needed in normal coding. Specifically, the developer must create an HP VEE definition file and a Unix shared library to go along with his functions.

The functions themselves can be written almost as they would be for any other development effort. Because HP VEE hides the developer from many programming details and

most functional languages do not, there are a few unique issues when they are used in STA. CLIB functions must dynamically allocate any additional memory they need (and deallocate it when they are done to prevent a memory leak). HP VEE handles memory for parameters passed to compiled functions, but if the compiled function needs to enlarge a parameter it must allocate the additional memory itself. Also, because HP VEE implicitly converts data types and functional languages usually do not, care must be taken to prevent type conflicts.

HP VEE requires a *definition file* to allow data sharing between HP VEE and the functional language. HP VEE determines the type of the data it should pass to and expect from the compiled function based on the contents of the definition file. The definition file defines the function's name, parameters and parameter types, and return type, and is syntactically similar to a C function declaration.

The compiled functions must be compiled as position-independent code and linked with a special option to construct a Unix *shared library.* This causes the linker to store relative addresses for entry points rather than absolute addresses. Storing relative addresses allows the shared library to be bound to the HP VEE environment at run time. When the library is bound, these relative entry points are converted to absolute addresses, which HP VEE uses to call the compiled functions. Once the CLIB is linked as a shared library and the definition file is written, HP VEE programs can call CLIB functions just like they call other dynamically-loaded library functions.

4.6 Naming and documentation

Without adequate documentation and sensible naming conventions, it is almost impossible to understand the code in a large system. Documentation and conventions are even more important when reuse of code is a primary goal. The naming conventions and documentation requirements for the STA system are described below.

4.6.1 Naming conventions

Naming conventions clarify the purpose of each library, function, and variable in the STA system. It is hoped that a developer will be able to identify the scope and purpose of any system component based solely on its name. If nothing else, the developer should be able to quickly determine where more detailed documentation can be found simply by looking at a component's name. Table 4.1 summarizes the naming conventions used in the STA system.

Named Element	Name and Description	Examples
Test file	*<tname>.test*: <tname> is the name of the test as found in the sequence file.	logon_operator voice_call
HP VEE / XRunner interface library	*<XRlibname>_IF*: <XRlibname> is the name of the XRunner library being mirrored.	gui_lib_IF cmdln_lib_IF
API library	*<cat>*: <cat> describes the category of functions found in this API.	call_processing data_processing
HIF library	*<dname>*: <dname> is the name of the hardware device the library acts on.	AM8e Fireberd Redcom_MDX
CLIB	*<libname>_clib* | *<name>_clib*: <libname> is the name of the library that the CLIB is associated with, otherwise, <name> is a descriptive name	am8e_clib Fireberd_clib database_ops_clib
Library function	*<libname>_<fname>*: <libname> is an optionally abbreviated library name. <fname> describes the individual function.	CP_dial() am8e_execute_cmds() Logon_logon() Logon_logoff() am8e_clib_check_cmds
library-specific global variable	*GL_<libname>_<vname>*: <libname> is the name of a library, <vname> is a descriptive name.	GL_am8e_output_len GL_am8e_clib_am8e_info
XRunner global array	*GL_<vname>_Array*: <vname> is a descriptive name.	GL_Menu_Array
other global variables	*GL_<vname>*: <vname> is a descriptive variable name	GL_HW_Types GL_HW_Config

Table 4.1 Summary of naming conventions

Each naming convention is based on slightly different reasons. Test files are located by STA by their name. Thus, test files must be based on the test name or the STA system will fail to find them.

HP VEE / XRunner interface libraries should be named the same as the XRunner libraries they mirror, followed by "_IF". This eliminates all confusion about which interface library mirrors which XRunner library. Also, the functions in the interface library should have the same names as those in the XRunner library (e.g. gui_lib_push_button and gui_lib_IF_push_button).

API libraries should be named after the group of functions they contain. All the functions in an API library should be related, and their common bond serves as the basis for the library name.

The names of the HIF libraries should simply be the name of the hardware they run on. While it is not necessary to include the full device name in the library name (e.g. am8e_PCM_VF_18-0035) major device types (e.g. am8a vs. am8e) should be reflected in the library name.

If a CLIB is used exclusively by one HP VEE library, the name of the CLIB should contain the name of the calling library to make this dependence visible. Compiled libraries that are used by multiple HP VEE libraries should be given names descriptive of their function. All compiled libraries should have "_clib" at the end of their name to clearly distinguish them from the interpreted HP VEE libraries.

The names of library functions should begin either with the library name or an acceptable abbreviation thereof. This makes it clear which library a function is from when a developer needs to view or edit its contents or description. If two HIF functions do the

same thing on a different type of hardware, or if two functions do the same thing on a different type of screen, the function names should be the same except for the library-name prefix. This will make function names more uniform and prevent naming conflicts between functions that would otherwise have the same name.

All global variables should be prefixed by "GL_". This clarifies which variables are global. Global variables generally fall into two different categories, those used exclusively within one library, and those used by multiple libraries. Global variables used within only one library should be prefixed by the library's name. Otherwise, they should just be given a descriptive name. To increase clarity, global arrays used in XRunner libraries are required to have a suffix of "_Array".

4.6.2 Documentation requirements

In order to ensure that additions to the STA system are as reusable as the original system, requirements for documentation have been developed. These fall into requirements for the documentation of new libraries and requirements for documentation of new library functions. Project-specific libraries and functions may have additional documentation requirements. They should, however, include at least the documentation required for additional generic libraries and functions.

Rather than creating a new library, many additions to the STA system will be done by adding functions to an existing library. This should be the case whenever a library's functionality can be logically extended to include the new function. Whether or not to create a new library is a common-sense decision left to the developer. The developer should look at the documentation of the existing libraries to decide whether it is more appropriate to

add a new library or augment an existing one. When creating a new XRunner library, a new XRunner / HP VEE interface library must also be created and documented.

Required library documentation

In addition to documentation of each library function, each library should be documented as a whole. The library documentation gives the developer an idea of the purpose and scope of the library. It also helps him determine when to add a new function to an existing library or when to create a new library. The minimal documentation required for each STA library is:

1. Name: Full name of the library.
2. Function prefix: The library name or an acceptable abbreviation thereof, used as a prefix for the names of all functions and global variables unique to this library.
3. Type: Type of the library, as described in section 4.5 (XRunner, HP VEE / XRunner interface, API, etc.).
4. Language: The language that the library is written in.
5. Functionality: Functionality provided by the library. This should be a high-level description of the type and purposes of functions found in the library. It should not be a list of the functions in the library.
6. Libraries called: A list of the libraries from which this library calls functions,
7. Other: Any other useful information about the library, including comments on its composition and member functions.

The generic documentation for generic libraries does not include the actual location (directory path) of the libraries-- this is project-specific. The project-specific documentation of generic libraries should state the location of the library in the directory tree and then reference the generic documentation.

Required function documentation

If a new function does not significantly change the character or functionality of the library it is being added to, then the documentation for the library itself does not need to be modified. While the library documentation might not need to change, the new function itself must always be documented.

The documentation discussed here is for individual functions, and is stored both in the function itself and in other locations. The following is a list of what should be included in the documentation for each library function. Any field that does not apply to a certain function can be left blank, and additional information can always be added.

1. Function: For XRunner functions and compiled functions, this is the name used to call the function from HP VEE. For HP VEE *User Functions*, this is the name as it would be seen in an HP VEE *Call Function* object used to call it. This should also list all the parameters with their respective types. For example:

   ```
   gui_lib_IF_push_screen( text scalar Screen_Name)
   ```

2. Library name: The full name of the library the function is found in.

3. Mirror function: For XRunner functions, this is the name of the HP VEE / XRunner interface function that mirrors it. For HP VEE / XRunner interface functions, this is the name of the XRunner function it mirrors.

4. Internal/External: Whether the function is intended to be internal or external to the library. Internal functions should not be used outside the library. External functions can, however, be used internally to promote code reuse within the libraries themselves.

5. Functionality: A description of what the function does. This should not include a discussion of the function's structure or implementation, but it may include suggestions for its use.

6. Entry requirements: The hardware and software set-up that the function needs to operate correctly. As examples, an am8e HIF function would require the presence of an am8e device (possibly in a certain configuration), and an XRunner GUI function might require that the SUT's GUI is present on the screen or has been defined in global variables.

7. Libraries used: The libraries that need to be in memory for this function to operate. Depending on the types of the libraries listed here, they should either be loaded in the test sequence where this function is called, or when the system is going through its initial hardware set-up (see section 4.3.3).

8. Global variables used: Global variables that the function uses. These can be HP VEE, XRunner, or other global variables, depending on the language in which the function was written. The list should describe which global variables the function needs defined or initialized (and how to do it), and which ones the function modifies.

9. Inputs: All inputs, along with their respective required types and shapes. If the inputs are further restricted (which is likely), then those restrictions should be listed here. Anything the test developer should know about the inputs that isn't described in the *Functionality* or *Entry requirements* sections should be listed here.

10. Outputs: Any outputs, along with their respective types and shapes. Also, the expected values and formats of the outputs should be described. This includes the meanings of return codes or error codes generated by the function.

11. Error states: A description of all the errors that this function might raise. These do not include errors passed along as return values from the function, only possible abnormal terminations of the function. They include HP VEE errors raised or exceptions that can't be handled by a CLIB function's language. A crash in a compiled function will crash the entire STA system and there is no way to avoid this, so any possible crashes in a CLIB function must be noted.

12. Composition: A brief description of the underlying implementation of the function. This should include a list of other functions that this function is dependent on.

13. Other comments: Any other comments or warnings that might be helpful to the test developer.

The method of documenting library functions depends on the language the function is written in. Functional languages allow the developer to store documentation as comments within the source code. HP VEE is not a functional language, but provides ways for the developer to put comments with any HP VEE object, known as the object's description. All library functions should be documented on paper, but they should also be documented within the function itself so the developer does not need to take the time to look through manuals while he is including library functions in his test.

Chapter 5
Conclusion and New Directions

This section will evaluate the design and performance of STA and discuss options for expansion of the system.

5.1 Design evaluation

Reusability

A major design goal for STA was to effectively reuse code, both within one application of the system and across different product lines. The STA system will be reusable for two reasons: the STA framework is generic, and many of the libraries are generic. The framework can be used on another product line without modification because it is independent of the product it is applied to. Any division that wants one will immediately have an automation framework for their regression tests. They will have to develop their own tests, but the development of the structure of the system need not be repeated. Because project-specific and generic functions have been separated into different libraries, these other product lines will also have all generic libraries available for their use. Thus they will already have a set of functions to assist in building their tests.

Expandability

STA's framework / library structure lends itself to easy expandability. Almost any expansions to the STA system can be put in new libraries or new library functions. Furthermore, adding libraries and functions will not make old tests incompatible with the new system. Tests will not need to be modified at all because they will use a subset of the new functionality. Thus expansions to STA will be backward-compatible.

Almost all of STA's functionality can be expanded. It can recognize new GUI screens by creating new global variables, XRunner libraries, and HP VEE / XRunner interface libraries. New external hardware types can be supported by adding new HIF libraries and modifying API libraries. New means of communicating with external hardware can be supported by adding communications libraries and modifying the Ports library. New levels of testing (unit testing, integration testing, etc.) can be supported simply by adding tests. If necessary, new generic libraries can be created to support other testing levels.

Ease of use

HP VEE was chosen as the primary language because (among other reasons) it makes test development easy. It is easier to learn than a functional language because writing an HP VEE program is very similar to writing a flow-chart. The data and control flows are visible on the screen and easy to modify because HP VEE language constructs relate to an intuitive concept of objects-- each HP VEE object has a name, well-defined functionality, and a visible location in the program. HP VEE also abstracts away many programming details (such as type conversions and memory allocation) from the developer.

Test development is also facilitated by libraries. A test developer will have common subroutines available when he begins writing his test. He can use them based solely on their specification, and need not worry about the details of their implementation.

STA directory tree

One aspect of the design may complicate the maintenance of the STA system. The project-specific elements of each instantiation of the system will be maintained by individual product lines, while the framework and generic libraries will be maintained by the

Software Tech group at HNS. This will be complicated by the results of a discussion about the directory tree structure that occurred during high-level design.

The choice was made to have the division between source languages used in the system at the highest level of the directory tree. The resulting directory structure is shown in Figure 5.1, and what I consider a better alternative is shown in Figure 5.2. The alternative structure divides between generic and project-specific code at the highest level in the tree. Because separate groups are maintaining the project-specific and generic code, the alternative directory structure would facilitate the incorporation of new releases of generic code. The tree shown in Figure 5.1 was chosen because XRunner places constraints on its directory structure and because the separation between languages is also a logical way to divide the highest level of the tree.

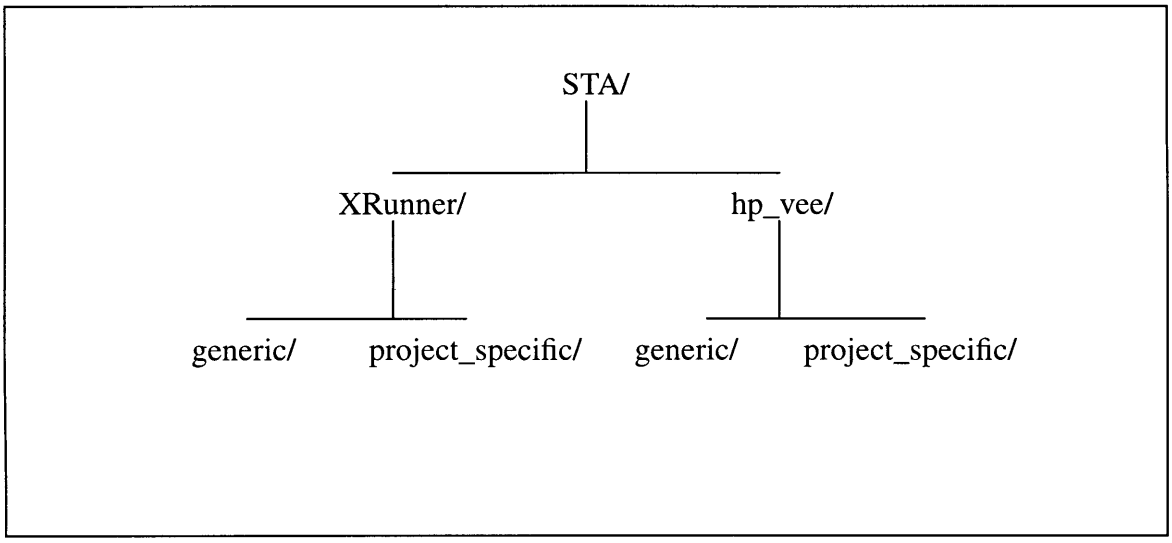

Figure 5.1 Existing TES STA directory structure

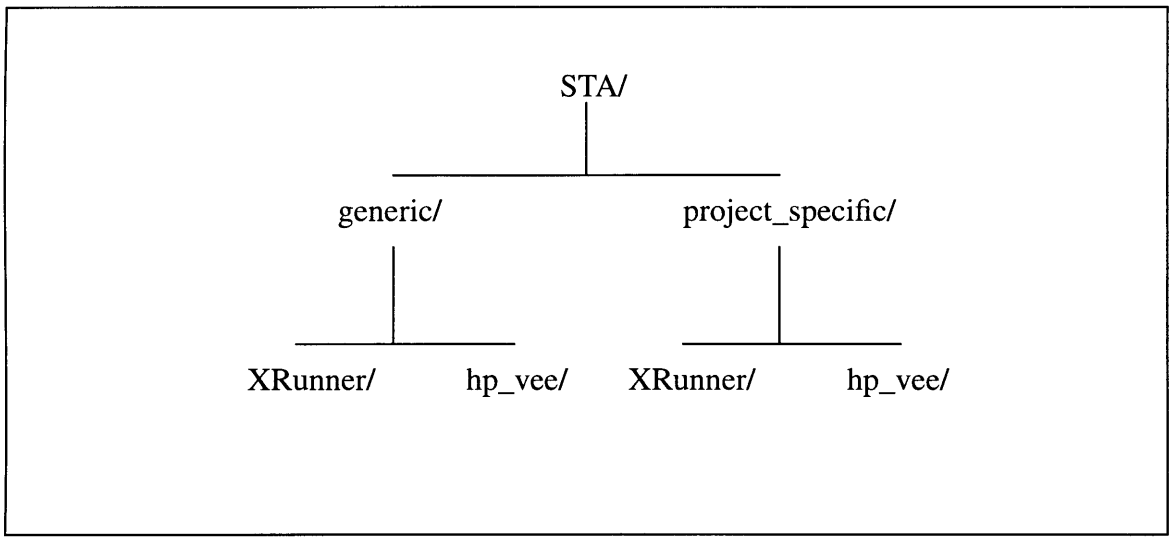

Figure 5.2 Proposed TES STA directory structure

5.2 Performance evaluation

The STA system was developed to improve the TES development cycle. While the system is still in the prototype stage, it has already begun to fulfil this goal.

As of March 25, 1996, the STA system had generated four software problem reports by running the prototype tests on the TES system. More defects have been uncovered by STA since then. Software problem reports are HNS's method for tracking faults in their products, so this is the equivalent of a developer finding four separate errors in the system. These problem reports were generated while running the prototype tests as regression tests. They were found while accessing a configuration screen on the TES operator interface GUI that is not regularly tested. The STA system discovered these errors earlier in the development cycle than they would have been discovered otherwise. This is the benefit of regression testing that was lacking before the use of the STA system. Early detection saves time and effort in the TES development cycle by detecting these errors before the release is distribtued either internally or externally.

As more tests are written for use with STA, the benefits will become more pronounced. The TES division can now do the weekly (or even daily) regression testing that it was unable to do before. This testing can be done overnight so it will not interfere with any other development activities. TES is now able to do regression testing because STA has increased the efficiency of the testing process.

5.3 New directions

The STA system has great potential to improve the TES development cycle, and is only at the beginning of its service life. Two expansions to the system are currently being pursued and a number of other possibilities exist.

Debapriya Sarkar, who was responsible for implementing all of the TES-specific tests and libraries, is continuing this work. Each test he writes is added to the regression testing sequence being run on the TES product. His goal is to automate 100% of the TES release tests and apply them as regression tests, and then to expand that suite to include tests that were never performed before.

Braeton Taylor, who was responsible for the framework and GUI libraries, is developing libraries to facilitate unit-level testing. He is also promoting the use of STA in other groups in HNS. Currently, developers construct "scaffolding" code during unit-level testing. This scaffolding may include stubs or test drivers for the unit they are developing, as well as other code. Braeton's new libraries will help them develop stubs and drivers faster. In addition, it is hoped that developers will be able to create project-specific unit-level testing libraries to encapsulate common subroutines in much the same way as other project-specific libraries captured common subroutines. Finally, there is the possibility that unit-level tests can be incorporated into the regression test sequences, making them even more thorough.

Developing more TES tests and the unit-level test libraries are currently the most promising directions for further STA development. There are, however, a number of other possibilities that can be explored in the future.

The TES channel unit (CU) hardware has a debugging port that allows developers to communicate directly with the hardware. This would be a very useful way to analyze the results of hardware tests and obtain CU status. In the future, TES-specific hardware libraries should be developed to allow test developers to communicate directly with the CUs.

The results analysis capabilities of STA are currently very limited. STA returns a listing of the tests run, their pass / fail status, return code, and a one-line text message to the STA *Test Results* screen and to a time-stamped file. As more tests are developed for STA, improved results analysis capabilities would be useful. Possibilities include comparison of multiple test runs, graphical representation of test results, and the generation of useful summary information.

The IEEE-488 bus is used by many types of automated test equipment. It would be prudent to add the capability to control external hardware through the IEEE-488 bus to the hardware communication libraries. This would increase the number of types of test equipment that could be controlled by STA.

Finally, the benefits of the STA system will grow as more divisions within HNS use it. They will be able to take the existing generic architecture and expand it with project-specific libraries. The same improvements to the TES development cycle can be applied to other HNS divisions once they begin using STA to automate their tests. As more divisions being using STA, the framework itself and the generic libraries can be reevaluated in a larger context, and improvements to their functionality and design that weren't seen before may become apparent.

5.4 Conclusion

The development of the STA system helps fill a gap in available test automation tools. While many tools exist for automating hardware tests or software tests, examples of tools that can be applied to either (and more importantly, be applied to both at the same time) are much more rare.

For large products, manual regression testing is not a realistic option because of time and manpower constraints. Without the proper tools, automated regression testing of combined hardware and software systems is impossible. The lack of adequate regression testing prevents errors from being discovered early, and increases the overall length of the system development cycle. The STA system solves this problem for the TES product, and is structured so it will be able to solve the same problem for different products.

After its initial application to regression testing, STA can be applied to other levels and types of testing. It can be used to assist developers in unit-level testing by providing standardized procedures for creating stubs and test drivers. This would allow them to do unit-level testing faster and possibly incorporate their unit-level tests into regression testing sequences. It can also be used to automate the final release tests, which would help shorten the typical (for TES) four-week release testing phase.

The structure of the STA system lends itself to reuse by separating generic and project-specific information and establishing documentation and naming requirements. Developing reusable code is one of the challenges of modern software engineering. Most code is applied to one product and does not contribute to the development of others. STA was designed so it could improve the development cycle of multiple product lines inside or outside of Hughes Network Systems.

References

[ABF90] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, New York, NY, 1990.

[Atl92] Joanne M. Atlee. Automated Analysis of Software Requirements. Thesis (Ph.D.) directed by Dept. of Computer Science, University of Maryland at College Park, 1992.

[Bez84] Boris Bezier. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Co., New York, NY, 1984.

[BFK85] Thomas Bracewell, Steven Feuchtbaum, and Scott Knowlton. TVL: A High Level Predictive Test Vector Language. In *Proceedings: Automated Testing for Electronics Manufacturing Conference, East*, p. IX5-IX9. Morgan-Grampian, Boston, MA, 1985.

[BL70] T. B. Brereton and W. J. Lukowski. An Automatic Test System For Software Checkout of Naval Tactical Data Systems. In *Proceedings of the Joint Conference on Automatic Test Systems, No. 17*, p. 375-385. IERE, London, 1970.

[CAB91] S. Chakradhar, V. Agrawal, and M. Bushnell. *Neural Models and Algorithms for Digital Testing*. Kluwer Academic, Boston, MA, 1991.

[Cla70] C. Clarke. A Computer Controlled Digital Equipment Test System. In *Proceedings of the Joint Conference on Automatic Test Systems, No. 17*, p. 147-148. IERE, London, 1970.

[Cli85] James E. Clicquennoi. High Volume Board Testing Techniques. In *Proceedings: Automated Testing for Electronic Manufacturing Conference, West*, p. IV17-IV21. Morgan-Grampian, Boston, MA, 1985.

[CW95] George T. Clemence and Sandra J. Walimaa. Lessons Learned on a Large Scale ATS Software Development Project. In *Conference Record: Autotestcon '95*, p. 118-126. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[Deu82] Michael S. Deutch. *Software Verification and Validation: Realistic Project Approaches*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1982.

[Dun84] Robert H. Dunn. *Software Defect Removal*. McGraw-Hill, New York, NY, 1984.

[GL91] Keith Brian Gallahger and James R. Lyle. Using Program Slicing in Software Maintenance. In *IEEE Transactions on Software Engineering, Vol. 17, No. 8*, p. 751-761. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[Hea81] James T. Healy. *Automatic Testing and Evaluation of Digital Integrated Circuits*. Reston Publishing Company, Inc., Reston, VA, 1981.

[Hei95] John E. Hesier. The Evolution of ABBET. In *Conference Record: Autotestcon '95*, p. 35-39. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[Her88] John Herbert. Temporal Abstraction of Digital Designs. In G. J. Milne, editor, *The Fusion of Hardware Design and Verification*, p. 1-24. North-Holland, Amsterdam, 1988.

[HNS1] John Vespoli. TES Automated Regression Testing Feature Requirements Specification. Hughes Network Systems internal document #8052250, 1995.

[HNS2] John Vespoli. Telephony Earth Station Automated Testing Project Plan. Hughes Network Systems internal document #8052022, 1995.

[HNS3] Debapriya Sarkar. TES Automated Regression Testing Feature Design. Hughes Network Systems internal document #1021332, 1995.

[HNS4] Braeton Taylor. System Test Automation Tester Guide. Hughes Network Systems internal document #8052733, 1996.

[HNS5] Braeton Taylor, Thayne Coffman, et. al. System Test Automation Developer Guide. Hughes Network Systems internal document #8052357, 1995.

[HNS6] Thayne Coffman. Automated Testing Generic Libraries Feature Design. Hughes Network Systems internal document #8052358, 1995.

[JK90] Niraj K. Jha and Sandip Kundu. *Testing and Reliable Design of CMOS Circuits*. Kluwer Academic, Boston, MA, 1990.

[KST95] F. C. M. Kuijstermans, M. Sachdev, and A. P. Thijssen. Defect-Oriented Test Methodology for Complex Mixed-Signal Circuits. In *The European Design and Test Conference*, p. 18-23. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[Lew92] Robert O. Lewis. Independent Verification and Validation: a Life Cycle Engineering Process for Quality Software. John Wiley & Sons, Ltd., New York, NY, 1992.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, 1986.

[Lon70] Lt. Col. R. W. A. Lonsdale. Management Planning for Multi-System Automatic Testing. In *Proceedings of the Joint Conference on Automatic Test Systems, No. 17*, p. 11-15. IERE, London, 1970.

[Mab85] George Mabry. Testing Digital VLSI Surface-Mount Technology. In *Proceedings: Automated Testing for Electronics Manufacturing Conference, East*, p. I24-I33. Morgan-Grampian, Boston, MA, 1985.

[May85] Lloyd N. Mayfield, Jr. Automated "End-of-the-Line" Testing with Multiple Test Stations Utilizing the "IEEE-488" Bus. In *Proceedings: Automated Testing for Electronics Manufacturing Conference, East*, p. VI23-VI29. Morgan-Grampian, Boston, MA, 1985.

[MJF95] Rajarshi Mukherjee, Jawahar Jain, and Masahiro Fujita. VERIFUL: VERIfication using FUnctional Learning. In *The European Design and Test Conference*, p. 444-448. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[NATO93]NATO Advanced Study. *Verification and Validation of Complex Systems: Human Factors Issues*. Springer-Verlag, New York, NY, 1993.

[NMH95] Bill Neblett, Frank Meade, and Gary Hardenburg. Test Object Reuse: A Technology Demonstration. In *Conference Record: Autotestcon '95*, p. 180-191. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[NS89] Paliath Narendran and Jonathan Stillman. Formal Verification of the Soebel Image Processing Chip. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, p. 92-106. Springer-Verlag, New York, NY, 1989.

[Par87] Kenneth P. Parker. *Integrating Design and Test: Using CAE Tools for ATE Programming. IEEE Computer Society Press*, Washington, D. C., 1987.

[PS89] Bill Pase and Mark Saaltink. Formal Verification in m-EVES. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, p. 268-296. Springer-Verlag, New York, NY, 1989.

[RH77] C. V. Ramamoorthy and S. F. Ho. Testing Large Software With Automated Software Evaluation Systems. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology, Volume 2*, p. 123-139, 145-150. Prentice-Hall Inc., Englewood Cliffs, NJ, 1977.

[SS95] John W. Shepherd and William R. Simpson. A View of the ABBET Upper Layers. In Conference Record: Autotestcon '95, p. 51-56. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[Stu77] Leon G. Stucki. New Directions in Automated Tools for Improving Software Quality. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology, Volume 2*, p. 80-85. Prentice-Hall Inc., Englewood Cliffs, NJ, 1977.

[TMR95] Kirk D. Thompson, Robert L. McGarvey, and Jeff Rajhel. ABBET Architecture. In *Conference Record: Autotestcon '95*, p. 41-50. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[Wei85] Warren Z. Weinstein. New Directions in Functional Testing: Beyond Simulator Based Programming. In *Proceedings: Automated Testing for Electronics Manufacturing Conference, East*, p. VIII10-VIII16. Morgan-Grampian, Boston, MA, 1985.

[Wes85] Todd Westerhoff. The Role of the Engineering Workstation in Test Program Development. In *Proceedings: Automated Testing For Electronics Manufacturing Conference, Northwest*, p. III27-III30. Morgan-Grampian, Boston, MA, 1985.

[WPT95] Gwendolyn H. Walton, J. H. Poore, and Carmen J. Trammel. Statistical Testing of Software Based on a Usage Model. In *Software-- Practice and Experience, Vol. 25(1)*, p. 97-108. John Wiley & Sons, Ltd., New York, NY, 1995.