

SAND REPORT

SAND 2005-2454

Unlimited Release

Printed June 2005

APIttest User Guide v1.0

William C. McLendon III, Sandia

Ron A. Oldfield, Sandia

Prepared by

Sandia National Laboratories

Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.



Sandia National Laboratories

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.doe.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



APItest User Guide

v1.0

William C. McLendon III and Ron A. Oldfield
Dept. 9223
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110
{wcmclen,raoldfi}@sandia.gov

Abstract

APItest is a portable testing framework developed at Sandia National Laboratories to address some of the development challenges inherent in distributed systems software for massively parallel processing (MPP) machines. In particular, the scale of today's MPPs (tens of thousands of processors) make it difficult for the developer to isolate and/or reproduce errors because it is difficult to determine the exact state of the system during a crash. For smaller systems, the analyst typically finds errors by investigating log files, but this process becomes time-prohibitive as the number of client processors number in the thousands. APItest provides the capability to isolate and test individual components before they are deployed into a large distributed system—allowing the component developer to investigate an individual component's response to expected inputs without worrying about the correctness of external components or the system as a whole.

Sandia National Laboratories developed the APItest software as part of the testing integration effort of the SciDAC Scalable Systems Software (SSS) project. This document is a user guide for the APItest software.

Acknowledgment

Thanks to Narayan Desai, Scott Jackson, and Thomas Naughton for providing excellent feedback and requests that have significantly impacted the development directions of this tool.

The format of this report is based on information found in [1].

Contents

Nomenclature	7
1 Introduction	9
2 Feature List	12
3 Installation	13
3.1 Prerequisites	13
3.2 Installing from Source	14
3.3 Installing from a Binary RPM	14
3.4 Rebuilding a Source RPM	15
4 Running APItest	16
5 Test Scripts	16
5.1 Test Status Codes	18
5.2 “cmd” Tests	18
5.3 “script” Tests	20
5.4 “sss” Tests	21
6 Batch Scripts	23
6.1 The <test> element	24
6.2 The <dep> element	24
6.3 The <parameter> element	26
7 Viewing Results	27
8 Conclusion	28
References	30

Appendix

A Command Line Options	31
B Example Scripts	33
B.1 Example “cmd” Test Scripts	33
B.2 Example “script” Test Scripts	34
B.3 Example “sss” Test Scripts	35
B.4 Example Batch Scripts	36
C Encoding Special Characters into XML Text Blocks	37
D How To Create New Test Types	39
D.1 The do_newType() Function	40
D.2 The do_cleanup_newType() Function	40
D.3 The do_kill_newType() Function	41
E Output File Formats	43
E.1 Naming Conventions	43
E.2 Output File for Tests	44

E.3	Output File for Batches	46
F	Selected Screenshots	49
G	Troubleshooting.....	57
H	List of Installed Files	59

Figures

1	System components that comprise the SSS software. The components communicate through a standard, well-defined, network interface.	10
2	Listing: Generic test file outline	16
3	Listing: CMD test file outline	19
4	Listing: SCRIPT test file outline	21
5	Listing: SSS test file outline	21
6	Listing: Batch file outline	23
7	Listing: BATCH1.apb	24
8	Listing: BATCH2.apb	25
9	Listing: BATCH3.apb	26
F.1	Screenshot : command line	49
F.2	Screenshot : Main	50
F.3	Screenshot : Test Executing	51
F.4	Screenshot : Batch Result	52
F.5	Screenshot : Test Result	53
F.6	Screenshot : Dependency Failure	54
F.7	Screenshot : XML Error Detection	55

Tables

1	Standard <test> Attributes	17
2	Test Status Codes	18
3	Standard <command> Attributes	19
4	<input> and <output> Attribute Values for cmd and script tests	20
A.1	Command line options for APItest.	31
E.2	testResult Element Attributes.....	44

Nomenclature

- .apt** Three letter file name extension for *test* files.
- .apb** Three letter file name extension for *batch* files.
- API** Application Programming Interface. In the context of APITest, API refers to the way in which a component interacts with the operating system and/or other components.
- CDATA** Character DATA is a feature of XML and HTML type documents. It is a text-string that exists between the start and close of an *element*. (i.e., `<element>CDATA</element>`).
- DOM** Document Object Model is a form of representation of structured documents (such as XML) as an object-oriented model. DOM is the official World Wide Web Consortium (W3C) standard for representing structured documents in a platform- and language-neutral manner. See http://en.wikipedia.org/wiki/Document_Object_Model for additional information.
- attribute** A feature of an XML document. An *attribute* is contained within an element, usually containing some meta-data to that element. (i.e., `<element attribute="">`)
- digraph** Directed graph.
- element** A feature of an XML document. An *element* in XML creates a new node of information. (i.e., `<element>`). These are sometimes referred to as *tags*.
- DAG** Directed Acyclic Graph.
- GUI** Graphical User Interface (pronounced “gooey”).
- SSS** Scalable System Software (A SciDAC project, details can be found at the following URL: <http://www.scidac.org/ScalableSystems/>).
- XML** eXtensible Markup Language is a simple, very flexible text format derived from SGML (*ISO 8879*). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. [<http://www.w3c.org/XML/>]. Also see [2].

APItest User Guide

v1.0

1 Introduction

Software testing and debugging has long been a challenge for parallel and distributed systems. Because of the lack of effective and portable tools, testing, even on small systems, is a tedious and time-consuming task, often requiring as much as two-thirds of the overall cost of software production [3]. On today's MPPs (tens of thousands of processors), testing and debugging is exacerbated by the fact that it is difficult to determine the exact state of the system during a crash, making it nearly impossible to isolate and/or reproduce errors. For small and mid-size systems, the analyst typically finds errors by investigating log files, but this process becomes time-prohibitive as the number of client processors number in the thousands. In this paper, we provide a brief description of systems software and describe a software package called “APItest” designed to help the developer overcome some of the challenges inherent in developing distributed systems software for massively parallel processing (MPP) machines.

Systems software for MPP systems consists of a variety of tools that perform services to manage the use of system resources by applications. These tools may include services to manage and launch jobs, track account usage, provide security and reliability, and so forth. Since each system and computing environment is different, the required tools and policies for usage may be entirely different for each system. To satisfy these requirements, existing systems software consists of (often proprietary) tools highly tuned to match the specific requirements of its particular system, creating a suite of tools that is not portable and has little value outside of their particular environment. To address this problem, the SciDAC Scalable Systems Software (SSS) project [4] was formed to leverage the systems-software development experience of the DOE laboratories [5, 6] to build an open-source, Linux-based, systems-software suite for large-scale commodity clusters.

In general, there are two types of architectures for systems software on MPP systems:

1. **Configurable Architecture**—the systems-software consists of a small set of tools that each contain a variety of functionality that the administrator configures to meet the needs of the system.
2. **Component Architecture**—the systems-software consists of a large set of tools that each provide specialized functionality. The administrator composes the system by

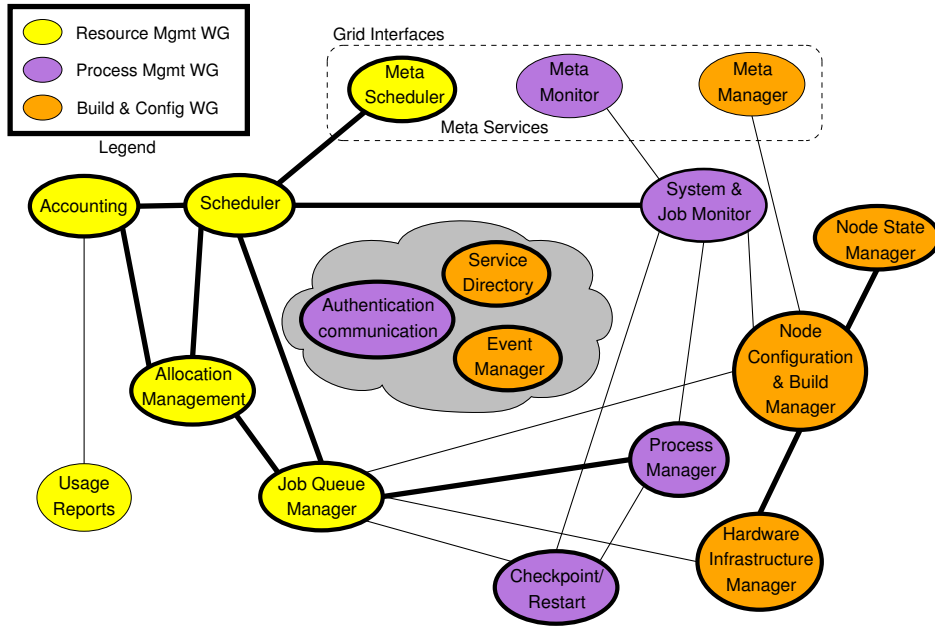


Figure 1. System components that comprise the SSS software. The components communicate through a standard, well-defined, network interface.

selecting tools appropriate for the system.

The members of the SSS project chose a *component* architecture (illustrated in Figure 1) for several reasons. In contrast to tools in a configurable architecture, a component has a small code base, making it less susceptible to error, more predictable with respect to performance, and easier to develop. For example, when designing a configurable tool, the developer has to anticipate the needs all current and future applications; however, the developer of a “lightweight” component only implements functionality needed by a single target system.

As leader of “Testing and Integration” working group of the SSS project, Sandia is responsible for finding or developing tools to validate the correctness of individual components in the SSS suite. While there are several existing tools for testing and validating systems software [7, 8, 9, 10, 11, 12, 13], none of these tools provide all of the features desired by the SSS group that specifically address the testing challenges of large-scale MPP systems. In particular, the SSS group wanted a portable testing tool that allowed devel-

opers to test the interface of the components in isolation, they wanted to define complex dependency relationships between successful tests, and they wanted a nice graphical user interface for interactive testing and a command-line interface for off-line testing. APItest implements each of these features. APItest provides isolation by using “black-box” testing that evaluates the response of an isolated component to various inputs. Isolation testing allows the developer or system administrator to evaluate the correctness of a component without worrying about the correctness of external components or the system as a whole. This feature enables the rapid development of multiple implementations of the same component as well as concurrent development of other system services because the developer does not have to rely on other working components to verify correctness or compliance with the standard network interface.

APItest also allows the user to provide arbitrary definitions of success, as opposed to the simple string matching provided by most testing software, and it can pass conditional tests based on statistical results. For example, the user can declare a test a successful if 40% of “sub-tests” succeed.

APItest is written entirely in Python, making it portable across many different operating systems. The user can run APItest from the command line or through a web-based graphical user interface. Although we developed APItest to evaluate systems software, APItest offers a flexible and extensible test driver framework for the validation and testing of a wide range of applications.

This document is a user guide for the APItest software. Section 2 provides a complete list of the features of APItest, Section 3 lists the prerequisites and describes how to install the software, Section 4 describes how to run APItest either from the command line or from a web-browser GUI, Section 5 describes how to develop test scripts for evaluating components, Section 6 describes how to batch tests together, and Section 7 describes how to view results. We also describe command-line options and provide a variety of examples in the appendix.

2 Feature List

Some of the features in APItest are:

- High portability
 - Written in Python for high portability.
 - Text only mode, perfect for running scheduled jobs.
 - Graphical mode via web browser for interactive use.
 - Safe mode for browser.
 - Open Source (LGPL)
- Scripts
 - Tests are scripted in XML.
 - Two types of scripts:
 - * Test scripts.
 - * Batch scripts.
 - Environment variables
 - SUID capability
 - Timeouts
 - Nested batch files
- Several built in test types.
 - Run embedded scripts.
 - Run commands via command-line.
 - Scalable System Software tests network API.
- Test Validation
 - Validate specific outputs – only check what matters.
 - Pattern matching support.
- Rich dependency system.
 - Tests within batches can be ordered.
 - Conditional execution based on dependencies.
 - Overall batch file status determination.
- Script error detection and reporting.
- Test execution orders can be specified.
- Easy to add new test types to APItest.
- Execution results archival on disk.

3 Installation

This section provides instructions on how to install APItest. There are packages that need to be installed on a system in order for APItest to work. The following subsections describe what prerequisites are needed and the installation procedure.

3.1 Prerequisites

APItest requires three packages to be installed before starting APItest. The following table provides a quick summary of the requirements for APItest and web URLs from which they can be obtained.

1. Python	≥ 2.2	Python runtime environment http://www.python.org
2. ZopeInterfaces	≥ 3.0	ZopeInterfaces (for Twisted) http://www.zope.org/Products/ZopeInterfaces
3. Twisted	≥ 2.0	Twisted application framework
4. TwistedWeb	≥ 0.5	Twisted Web framework http://www.twistedmatrix.com
5. ElementTree	≥ 1.1	ElementTree module for Python http://effbot.org/downloads/#elementtree

Python is the programming language that APItest is developed in. Since Python is an interpreted language kind of like Perl, it must be installed or APItest cannot run.

ZopeInterfaces is a prerequisite to the Twisted Framework. It must be installed prior to installation of Twisted and TwistedWeb.

Twisted is a framework, written in Python, for writing networked applications. It includes implementations of many useful network services such as a web server, etc. TwistedWeb is part of the Twisted framework but has recently been partitioned from the main distribution into its own package.

ElementTree is a Python library which parses XML into a DOM tree. It provides a much cleaner and easier to use interface than the default XML parser that comes packaged with Python. We make use of this for loading and processing test scripts and in saving results.

3.2 Installing from Source

The following steps outline the procedure for installing APItest after the prerequisite packages (Sec. 3.1) have been installed.

1. Make sure all prerequisites have been installed.
2. Extract the tar.gz archive.

```
$ tar -xzf apitest-1.0.tar.gz
```
3. CD into the directory created during extraction.

```
$ cd apitest-1.0
```
4. Build and install into the **default** directory (/usr/local/apitest).

```
$ make install
```
5. Build and install APItest into a **user-specified** directory.

```
$ make install PREFIX=installation_path
```

The installation script installs the `libapitest` module into the `site-packages` directory of your current python installation. This module is used by APItest to perform its functions. The APItest executables are installed into `/usr/local/apitest` by default, but this can be overridden by providing the `PREFIX` parameter to the `make` command as shown in step 5.

3.3 Installing from a Binary RPM

Installing APItest from a binary RPM on an x86 computer can be accomplished using the following steps on a PC running Linux after the required prerequisites have been installed.

1.

```
$ rpm -Uvh apitest-1.0.i386.rpm
```

See Appendix H for a complete listing of the files this will install. The listing may also be obtained by issuing the `rpm` command:

```
$ rpm -qpl apitest-1.0.i386.rpm
```

.

3.4 Rebuilding a Source RPM

The following steps will guide you thorough building and installing from a source rpm.

1. Create a binary RPM.

```
$ rpmbuild --rebuild apitest-1.0.src.rpm
```

2. Copy the binary RPM to your home directory.

```
$ cp /usr/src/redhat/RPMS/i386/apitest-1.0.i386.rpm /.
```

3. Install APItest from the binary rpm.

```
$ rpm -Uvh apitest-1.0.i386.rpm
```

4 Running APItest

APItest can be run either via the command line only or it can be run with a web-browser based GUI. The former mode allows APItest to be run as a batch or system scheduled task, while the latter allows a more interactive mode of execution. A full listing of allowable command line options is available in Appendix A, table A. Some example sets of command line options are:

Help	- \$ apitest -help
Test-Only	- \$ apitest [options] -f <i>input_file</i>
Graphical	- \$ apitest [options] httpd [httpd_options]
Graphical Help	- \$ apitest httpd -help
View-Only	- \$ apitest httpd -viewonly

5 Test Scripts

Tests are written in XML text files. We refer to these files as *scripts*. Currently there are two types of scripts that APItest recognizes, *test* and *batch* scripts. A *test* script instructs APItest to execute a command or task. The basic XML structure of a test file is shown in Figure 2.

```
<testDef>
  <info>CDATA</info>
  <test type="type_name" attributes>
    <input name="input_name">CDATA</input>
    :
    <output name="output_name" format="format">CDATA</output>
    :
  </test>
</testDef>
```

Figure 2. Listing: Generic test file outline

The top-level root element is `<testDef>`. It serves as the root level element for the XML document. This element contains two other elements, an `<info>` element and a `<test>` element.

The `<info>` element is common to ALL APItest input files. There are no attributes associated with this element. The purpose of this element is for the test developer to write notes or comments in. Otherwise, this element is not used by APItest for any actual testing. For that, we use the `<test>` element.

The `<test>` element contains everything that APItest needs to know in order to execute a test. Table 1 shows the attributes associated with this element. The most critical attribute for this element is the *type* attribute. It tells APItest which handler to use to run this test. Without it, the test will break and cause unpredictable behavior in APItest. Currently, APItest comes with three predefined test *types*: *cmd*, *script*, and *sss*. We say a test is a “cmd test” if the type attribute of `<test>` equals “cmd”.

Other parameters, such as working directory, timeout, or matching expectation can be controlled via the optional attributes. These are also listed in table 1. There are also some sub-elements that can be contained within `<test>`.

APItest will look for `<input>` and `<output>` within a `<test>` element. There can many or none of these, as needed by a specific test. Their names suggest their function in that an `<input>` element provides *inputs* to the test and `<output>` elements specify the *expected outputs* of this test. Since `<input>` and `<output>` attribute values are somewhat dependent on what kind of test they’re being used in, we will describe them in more detail in later sections.

<test> Element Attributes						
Optional	Attribute	default	Description	cmd	script	sss
No	type		Type of test to run. (REQUIRED)	Y	Y	Y
Yes	timeout	-1	Timeout in seconds (-1 = infinite).	Y	Y	Y
Yes	match	YES	PASS if actual output matches expected? (YES/NO)	Y	Y	Y

Table 1. Standard `<test>` Attributes

5.1 Test Status Codes

A status code is the final exit status of a test. For instance, if a test matched all of its expected outputs then we might say that the test PASSED. Table 2 provides a listing of status codes and a brief description of each.

Test Status Codes	
PASS	The test passed
FAIL	The test failed
FAILDEP	The test did not execute because of one or more failed dependencies.
TIMEOUT	The test ran too long and was killed by APItest

Table 2. Test Status Codes

5.2 “cmd” Tests

A *cmd* test is a test that executes some command via a direct command-line call. To specify a test as a “cmd” test, the `type` attribute in `<test>` should be “cmd”, or rather:

```
<test type="cmd">
```

These tests are designed to run some other preexisting binary or executable script on the system. These tests require one additional XML element to be specified inside `<test>`, called `<command>`.

The `<command>` element is used to specify what the actual command we are executing is. For instance:

```
<command>ls</command>
```

will instruct APItest to execute the UNIX directory listing command “ls”. Building on the basic test file structure, we can now see the general structure of a “cmd” test in figure 3.

There are several attributes we can specify for the `<command>` attribute that affect how and where the test is run. We can provide a particular working directory or run a command under a different user id. We can also specify which shell to run the command from such as *bash* or *csh*. Table 3 provides a listing of the attributes along with brief descriptions.

```

<testDef>
  <info>CDATA</info>
  <test type="cmd" attributes>
    <command options>exec</command>
    <input name="input_name">CDATA</input>
    :
    <output name="output_name" format="format">CDATA</output>
    :
  </test>
</testDef>

```

Figure 3. Listing: CMD test file outline

<command> Element Attributes						
Optional	Attribute	default	Description	cmd	script	sss
Yes	interpreter	/bin/sh	Interpreter for test.	Y	Y	
Yes	uname	current	User name to execute the command Requires <code>root</code> permission.	Y	Y	N
Yes	wdir	/tmp/	Working directory.	Y	Y	

Table 3. Standard <command> Attributes

After the <command> element, we can optionally add some <input> and <output> elements. Table 4 shows the possible attributes for these elements for *cmd* and *script* tests. Input elements can provide command-line arguments and stdin buffers to the commands being execute. It is allowed to specify multiple arguments.

APItest will recognize only stdout, stderr, or status in output elements for *cmd* and *script* tests. These values are the only ones that will make sense for a script or command since they typically write to standard output, standard error, and set an exit status upon completion. We also allow two different types of expected output buffers: regular expressions (regex) and string literals (literal).

If a regular expression is provided, APItest will determine if the actual output matches the expected regular expression. If the expected output is a string literal, APItest will do a direct string comparison.

If the test developer wishes to ignore some particular output stream, such as the standard error buffer, they can omit a `<output name="stderr">` element and APItest will ignore standard error. A good rule of thumb here is that APItest will only check what it's told to, everything else is ignored.

<input> Attribute Values	
name="argument"	Specifies an argument to the command.
name="stdin"	Specifies a string to send into the stdin buffer
<output> Attribute Values	
name="stdout"	Specifies this is the expected <i>stdout</i> buffer.
name="stderr"	Specifies this is the expected <i>stderr</i> buffer.
name="status"	Specifies this is the expected <i>exit status</i> .
format="literal"	Expected output is a literal string.
format="regex"	Expected output is a regular expression.

Table 4. `<input>` and `<output>` Attribute Values for `cmd` and `script` tests

5.3 “script” Tests

Script tests are similar to command tests. They execute a task on your system as specified by a script written in-line with the APItest test file. Figure 4 shows the outline for a script test. These are nearly identical to `cmd` tests, the only differences being that the test type attribute is set to “script” and the script body is placed in the CDATA buffer of the `<command>` element.

There are no additional attributes for the `<test>` element in a script test. See Table 1 for a listing of attributes for the `<test>` element in script tests.

Script tests also share the same attributes for the `<command>` element. Table 3 shows the relevant attributes. When running a script test, APItest will create and execute the script in the working directory provided by the test.

Finally, script tests share the same `<input>` and `<output>` format and attributes as `cmd` tests. This is reasonable considering the similar inputs and outputs a `cmd` or `script` will receive/produce.

```

<testDef>
  <info>CDATA</info>
  <test type="script" attributes>
    <command options>
      script body
    </command>
    <input name="input_name">CDATA</input>
    :
    <output name="output_name" format="format">CDATA</output>
    :
  </test>
</testDef>

```

Figure 4. Listing: SCRIPT test file outline

5.4 “sss” Tests

The third test type provided by APItest is the “sss” test. A sss test is designed to work with the *ssslib* communication package, which is part of the Scalable Systems project (<http://www.scidac.org/ScalableSystems>). These tests are used to test out APIs of system software components for this project.

```

<testDef>
  <info>CDATA</info>
  <test type="sss" destination="service_name" attributes>
    <input name="sendbuf">CDATA</input>
    <output name="recvbuf" format="format">CDATA</output>
  </test>
</testDef>

```

Figure 5. Listing: SSS test file outline

A sss test represents a single transaction with a SciDAC SSS aware application. A transaction consists of transmitting some buffer to a service via the *sss.ssslib* module. One caveat, we will need the Service Directory (SD) to be running on the system for the test

to work correctly. Also, we generally expect a transaction to consist of a message to a sss component and a response back.

Figure 5 shows the outline of a sss test. The basic outline is the same, but there are some differences. The `<test>` element requires a special attribute, `destination`, which specifies the destination service for the message. The communication library will lookup the destination in the service directory and transmit the message buffer to the correct component.

There is also a change in the `<input>` and the `<output>` elements. We only need one of each for a sss test. The `<input>` element specifies the send-buffer for a transaction. It requires the name attribute to be set to “sendbuf”. The CDATA is sent to the destination during the test via the ssslib communication library. The `<output>` element requires its name attribute to be assigned “recvbuf” to specify this as a receive buffer. If `<output>` is omitted APItest will do the send, but not wait for the receive. We do not recommend extensive use of this feature.

Please see Appendix B.3 for example SSS test scripts. Additional scripts are located in the `samples/scidac_sss/` directory.

6 Batch Scripts

A batch script is a script that does not run tests directly, but rather provides a listing of other tests that are to be run. Batch scripts can contain lists of tests, lists of other batches, or a mix of both. Batch scripts also allow dependencies to be set between tests to enforce execution order. These dependencies can also be set to allow a test to be executed conditionally depending on whether or not other tests passed or failed.

Batch files can PASS or FAIL in a similar manner to test files. By default, all tests in a batch file must pass in order for the batch file itself to PASS. A test can be removed from consideration by setting the `mustPass` attribute in a `<test>` elements, or by changing the default setting of `mustPass` in a `<parameter>` element.

```
<testBatch>
  <info>CDATA</info>
  <parameter key="mustPass" value="{ 'true'|'false' } />
  <test name="filename" mustPass="{ true |'false' } />
  :
  <dep parent="filename" child="filename" attributes/>
  :
</testBatch>
```

Figure 6. Listing: Batch file outline

This is the structure of a batch test. `<test>` elements do not have to precede `<dep>` elements. They can be placed in any order in the file.

The basic structure of a batch script is shown in Figure 6. The root element of a batch test is the `<testBatch>` element. It can contain an `<info>` element, a `<parameter>`, and any combination of `<test>` and `<dep>` elements.

6.1 The <test> element

<test> elements specify a test to be included in this batch file. The format of a <test> element is the following:

```
<test name="filename" [mustPass="True"|"False"] />
```

The `name` attribute is a test or batch filename that should be loaded and run by this batch. `mustPass` is an optional attribute that is used to tell APItest if this test is required to PASS for the batch file to PASS or not. `mustPass` is true by default, but can be changed by use of the <parameter> element.

```
<testBatch>
  <info>Some information about this file</info>
  <test name="A.apb"/>
  <test name="B.apb"/>
  <test name="C.apb"/>
  <test name="D.apb"/>
</testBatch>
```

Figure 7. Listing: BATCH1.apb

This listing shows a batch script that will launch tests A - D (note the .apb extension indicates these are tests). In this example, the tests will be run in no particular order.

6.2 The <dep> element

We add dependencies by using the <dep> element. Tests listed in a `dep` element do not have to be listed previously by a <test> element. The format of a <dep> element is:

```
<dep parent="t1" child="t2" [status="expected_status"]/>
```

Adding a dependency between two tests instructs APItest to run the parent (t1) before the child (t2). The **status** attribute is an optional attribute. It may contain PASS, FAIL, ANY, or MUSTRUN with PASS being the default value.

Setting status to PASS means that the parent must run and PASS or the child test will not be executed.

Setting status to FAIL means that the parent must run and FAIL or the child test will not be executed.

Setting status to ANY means that the child will not be run until after the parent is run. The child test will run regardless of the status of the parent.

Setting status to MUSTRUN means that the parent must run and either PASS or FAIL or the child test will not be executed.

If a test fails its dependency, it will not execute. APItest will assign the status **FAILDEP** to that test to indicate that it wasn't run because it did not meet all its dependencies.

Figure 8 extends our previous example shown in Figure 7 by adding dependencies.

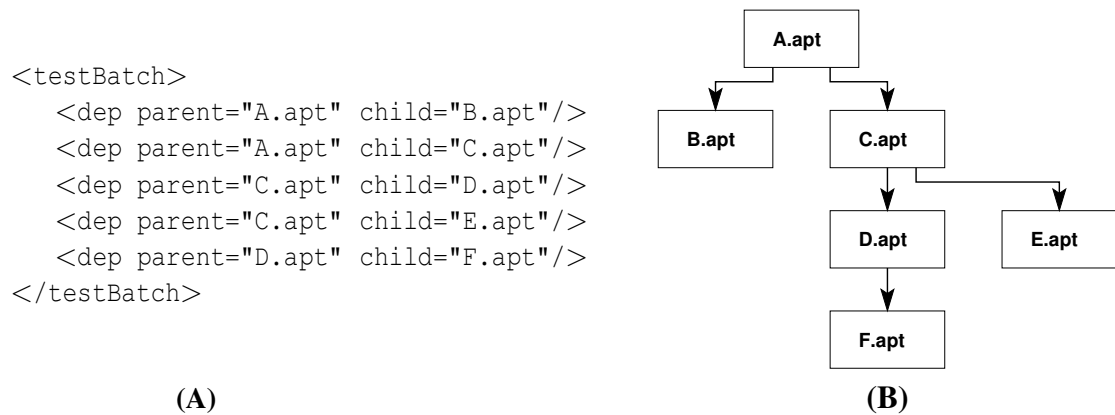


Figure 8. Listing: BATCH2.apb

(A) is a batch containing the tests listed in fig 7, with an ordering now specified. In (B) we see a graphical illustration of the tests showing the dependency hierarchy.

Figure 9 further extends our example by adding a status condition to the edge between A.apb and B.apb. In this case, test B.apb will execute if A.apb FAILED when it was run.

```
<testBatch>
  <dep parent="A.apb" child="B.apb" status="FAIL"/>
  <dep parent="A.apb" child="C.apb"/>
  <dep parent="C.apb" child="D.apb"/>
  <dep parent="C.apb" child="E.apb"/>
  <dep parent="D.apb" child="F.apb"/>
</testBatch>
```

Figure 9. Listing: BATCH3.apb

A third batch listing, illustrating setting an edge dependency. In this case, we add a restriction that B.apb will only run if test A.apb had a status of FAIL.

6.3 The `<parameter>` element

The `<parameter>` element is optional and has two attributes, *key* and *value*. Setting `<parameter key="mustPass" value="false">` will change the mustPass default setting to be *false* for all tests in this batch file. The value of mustPass is *true* by default.

Figure 7 shows an example batch script file which will run several tests. This file shows only `<test>` elements which may execute in any particular order. In practice they will usually execute in the order of appearance, but without explicit dependencies set we will not guarantee the ordering.

7 Viewing Results

There are three primary methods that users can view test results generated by APItest. The first method is to use the graphical mode of APItest to view results of a test that was run during the same session of APItest. The list of runs can be found by clicking on the **Browse Session Results** button.

These results will no longer be viewable once APItest is shut off. A user might still want to view these results graphically, so we have provided an offline browser as a part of APItest. To use this feature, simply start APItest with graphical mode enabled (`$ apitest httpd`) and click the **Saved** tab in the menu bar.

There is also a “view only” mode available for APItest which allows browsing the saved results but does not allow execution of any tests via the browser. This mode is enabled by adding `--viewonly` to the command line.

Finally, the results are saved on disk in text files in XML format. The raw output can be viewed in any text viewer a user wishes to use. These files are saved according to a particular naming convention. More information can be found on the naming of these files in Appendix E.1. There are some screen shots showing the result browser in Appendix F

8 Conclusion

APItest is a new open-source framework for driving application tests. It provides a portable and easy to use test framework due to its development in Python and use of XML scripting for test writing.

The initial design of APItest was to provide a capability to test the API of networked components such as those in cluster system software in order to validate their APIs. APItest allows interfaces to be tested for SciDAC Scalable System Software components using the ssslib package.

We also added the capability to APItest to run scripts as well as execute programs via a command line type shell. This gives users the capability to test virtually anything using APItest.

Finally, due to the object-oriented design of APItest, entirely new test types can be defined without having to significantly modify the APItest code. Appendix E.2 provides more detailed instructions for creating new test types. This allows APItest a large degree of customization for specific test environments while allowing tests to still be run natively under APItest.

References

- [1] T. K. Locke, “Guide to preparing SAND reports,” Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, Technical report SAND98-0730, May 1998.
- [2] “XML definition,” Internet. [Online]. Available: <http://en.wikipedia.org/wiki/XML>
- [3] R. Pressman, *Software Engineering, A Practitioner’s Approach*. McGraw-Hill, 1987.
- [4] A. Geist, “Executive summary of the Scalable Systems Software (SSS) project,” Internet. [Online]. Available: <http://www.scidac.org/ScalableSystems/Systems-2-pager.pdf>
- [5] D. S. Greenberg, R. Brightwell, L. A. Fisk, A. B. Maccabe, and R. Riesen, “A system software architecture for high-end computing,” in *Proceedings of SC97: High Performance Networking and Computing*. San Jose, California: ACM Press, Nov. 1997, pp. 1–15. [Online]. Available: <http://doi.acm.org/10.1145/509593.509646>
- [6] R. Evard, N. Desai, J.-P. Navarro, and D. Nurmi, “Clusters as large-scale development facilities,” in *CLUSTER ’02: Proceedings of the IEEE International Conference on Cluster Computing*. Washington, DC, USA: IEEE Computer Society, 2002, p. 54.
- [7] S. Ghosh and A. Mathur, “Issues in testing distributed component-based systems,” 1999. [Online]. Available: citeseer.ist.psu.edu/ghosh99issues.html
- [8] M. Harrold, D. Liang, and S. Sinha, “An approach to analyzing and testing component-based systems,” 1999. [Online]. Available: citeseer.ist.psu.edu/harrold99approach.html
- [9] S. E. Virginia, “A framework for practical, automated black-box testing of component-based software.” [Online]. Available: citeseer.ist.psu.edu/449872.html
- [10] S. Krishnamurthy and A. Mathur, “the estimation of reliability of a software system using reliabilities of its components,” 1997. [Online]. Available: citeseer.ist.psu.edu/krishnamurthy97estimation.html
- [11] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy, “A distributed resource management architecture that supports advance reservations and co-allocation,” in *Proceedings of the International Workshop on Quality of Service*, 1999. [Online]. Available: citeseer.ist.psu.edu/foster99distributed.html

- [12] S. H. Edwards, “Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential,” *Software Testing, Verification & Reliability*, vol. 10, no. 4, pp. 249–262, 2000. [Online]. Available: citeseer.ist.psu.edu/448240.html
- [13] A. D. Brucker and B. Wolff, “Testing distributed component based systems using UML/OCL,” in *Informatik 2001*, ser. Tagungsband der GI/ÖCG Jahrestagung, vol. 1, no. 157, Nov. 2001, pp. 608–614. [Online]. Available: citeseer.ist.psu.edu/brucker01testing.html
- [14] M. Wolf and C. Wicksteed, “Date and time formats,” September 1997. [Online]. Available: <http://www.w3.org/TR/NOTE-datetime>

A Command Line Options

General Command Line Options			
Short	Long	Default	Description
-d	--debug		Run APItest in debug mode.
-o	--oroot	./output	Output directory.
-v	--verbose		Verbose output.
-t	--timeout	-1	Timeout (seconds) to shut down APItest. (-1 = no timeout)
-T	--transient		Do not save output and results to disk.
-D	--sqldb	Disabled	Toggle data saving to a MySQL Database.
-P	--sqlpw	""	MySQL Database PW.
	--sqlreset		Resets MySQL tables IF -D and -P are correct
	--version		Print out twisted version information and exit.

Options For Text-Only Mode			
Short	Long	Default	Description
-f	--file		Input file (can be a .apt or a .apb file).

Options For Graphical Mode (httpd)			
Short	Long	Default	Description
-i	--iroot	./	Sets input directory for test files.
-h	--host	localhost	HTTP host URL. (i.e., http://host/)
-p	--port	2112	HTTP port number. (i.e., http://host:port/)
-w	--viewonly		View only mode. Only views saved results. Doesn't allow executing any tests via the browser.
-C	--nocss		Compatibility mode. Disables CSS stylesheet menus. Replaces the tabbed menu with a list-style menu. This might help with some browsers.

Table A.1. Command line options for APItest.

B Example Scripts

B.1 Example “cmd” Test Scripts

The following is an example *cmd* script that runs the UNIX command “ls -ltr” in /tmp/. The output elements specify that APItest will check the stdout buffer via the regular expression “.*”, which matches any output. This test will also validate that the command returns nothing to stderr and exits with a status of 0.

Example #1

```
<testDef>
  <test type="cmd" wdir="/tmp">
    <command interpreter="/bin/csh" wdir="/tmp">ls</command>
  </test>
</testDef>
```

Example #2

```
<testDef>
  <info>This test will list out a directory listing.</info>
  <shortDescription>Directory Listing</shortDescription>
  <test type="cmd">
    <command interpreter="/bin/csh" wdir="/tmp">ls</command>
    <input name="argument">-l</input>
    <output name="stdout" format="regexp">.*</output>
    <output name="stderr" format="literal" />
    <output name="status">0</output>
  </test>
</testDef>
```

Example #3

```
<testDef>
  <info>This test shows use of environment variables.</info>
  <shortDescription>Envvar Test</shortDescription>
  <test type="cmd">
    <command interpreter="/bin/csh">env</command>
    <input name="envvar" key="TEST_ENVVAR_1">foo</input>
    <output name="stdout" format="regexp">.*(TEST_ENVVAR_1=foo).*</output>
  </test>
</testDef>
```

B.2 Example“script” Test Scripts

In this case, the script prints the characters ‘a’, ‘b’, and ‘c’ each on a separate line. The *interpreter* attribute in the command element specifies which interpreter this script is run as, and the script will be executed from the wdir directory. We also left out an output element specifying stderr, which tells APITest to completely ignore any output to stdout.

Example #1

```
<testDef>
  <info>Runs a simple script.</info>
  <shortDescription>Script Test</shortDescription>
  <test type="script">
    <command interpreter="/bin/csh" wdir="/tmp/">
      foreach i ('a' 'b' 'c')
        echo "$i"
      end
    </command>
    <output name="stdout" format="regex">a\nb\nc\n</output>
    <output name="status">0</output>
  </test>
</testDef>
```

Example # 2

```
<testDef>
  <info>Script that prints out a time stamp.</info>
  <shortDescription>Script Time stamp</shortDescription>
  <test type="script">
    <command interpreter="/bin/csh" wdir="/tmp/">
      foreach i (1 2 3 4 5 6 7 8 9 10)
        set theDate = `date`
        echo "timestamp: ${i}: $theDate"
        sleep 1
      end
    </command>
    <output name="stdout" format="regex">(. *timestamp.*){10}</output>
  </test>
</testDef>
```

B.3 Example “sss” Test Scripts

An example SSSlib test which transmits a buffer to the service directory and expects any kind of output in return.

```
<testDef>
  <test type="sss" destination="service-directory">
    <shortDescription>SSS Test</shortDescription>
    <input name="sendbuf">&lt;get-location&gt;&lg;location
component='service-directory' host='*' port='*' protocol='*'
schema_version='*' tier='*'/&gt;&lt;/get-location&gt;</input>
    <output name="recvbuf" format="regex">.*</output>
  </test>
</testDef>
```

B.4 Example Batch Scripts

Here we have some examples of some batch scripts:

Example #1

```
<testBatch>
  <info>Sample batch script with no dependencies</info>
  <shortDescription>Batch #1</shortDescription>
  <test name="samples/cmd/cmd_test_1.apb"/>
  <test name="samples/cmd/cmd_test_2.apb"/>
  <test name="samples/cmd/cmd_test_3.apb" mustPass="false"/>
</testBatch>
```

Example #2

```
<testBatch>
  <info>Sample batch script with dependencies</info>
  <parameter key="mustPass" value="False"/>
  <test name="samples/cmd/cmd_test_1.apb" mustPass="True"/>
  <dep parent="samples/cmd/cmd_test_1.apb"
    child="samples/cmd/cmd_test_2.apb"/>
  <dep parent="samples/cmd/cmd_test_1.apb"
    child="samples/cmd/cmd_test_3.apb" status="PASS"/>
  <dep parent="samples/cmd/cmd_test_3.apb"
    child="samples/cmd/cmd_test_2.apb" status="ANY"/>
</testBatch>
```

Example #3

```
<testBatch>
  <dep parent="samples/cmd/cmd_test_1.apb"
    child="samples/cmd/cmd_test_2.apb"/>
  <dep parent="samples/cmd/cmd_test_1.apb"
    child="samples/cmd/cmd_test_3.apb" status="PASS"/>
  <test name="samples/cmd/cmd.apb"/>
  <dep parent="samples/cmd/cmd_test_3.apb"
    child="samples/cmd/cmd_test_2.apb" status="ANY"/>
  <dep parent="samples/cmd/cmd_notfound_1.apb"
    child="samples/cmd/cmd_test_4.apb" status="FAIL"/>
</testBatch>
```

C Encoding Special Characters into XML Text Blocks

One problem encountered in writing tests, especially for SciDAC SSS components (which transmit XML encoded messages from one component to another) is how exactly we can encode an XML message into the CDATA portion of another XML script without confusing the parser. For example, if we want to put the text “<send_data>test buffer</send_data>” into a <buffer> element, we might try the following:

```
<buffer><send_data>test buffer</send_data></buffer>
```

Unfortunately, this will confuse an XML parser because it will interpret the “<” character in <send_data> as the start of a new element. The way to get around this is to use a “<” in place of “<” in the CDATA buffer. Our example will work if we make it look like this:

```
<buffer>&lt;send_data>test buffer&lt;/send_data></buffer>
```

This change will allow the XML message to be encoded within an APItest script file. We can also use “>” to replace “>” and “&” to replace “&” characters too.

D How To Create New Test Types

Developing a general test framework is no easy task because every application is different and every development environment is also unique. Our solution to this problem is to develop APItest in an object-oriented manner and provide an interface from a test handler into the framework itself that is easy for a developer to use.

We provide several test handlers with APItest already (cmd, script, and sss test types). Extending these is not difficult for basic tests. The basic procedure is as follows:

1. Obtain the source distribution (the .tar.gz file) and extract it.
2. Edit the `testHandler.py` file in the `libapitest/` directory. For a new test type, `newType`, add the following function definitions to the `testHandler` class.

- `do_newType`
- `cleanup_newType`
- `kill_newType`

3. Install your modified version:

```
$ python setup.py install --install-data=PREFIX
```

Once this is finished, we would like to run our new test type. This is done via the `<test>` element in a .apt file. One might look like: `<test type="newType">`.

The following sections will explain what each of the three functions are and what APItest expects from each.

D.1 The `do_newType()` Function

This function is the workhorse of a test. It is responsible for executing the test and returning the results back to the APItest calling framework. Since APItest is built on the Twisted framework, we don't use a stack-based call system, rather we use callbacks for tests. This allows the web-browser and other handlers to perform their functions while a test is still running.

The callback is executed via inserting this command before exiting:

```
reactor.callLater(0.0, self.procReturned, rval)
```

This tells the Twisted reactor to call the function `self.procReturned(rval)` 0.0 seconds after the currently running function exits. `rval` is a Python *dictionary* object storing *key:value* pairs. Each key corresponds to the an `<output name="key">` element, and the *value* stores the *actual result*. APItest takes `rval` and compares the *key:value* pairs with the expected results for a test to determine if the test passes or fails. This is the minimal requirement for a test to return to the framework, though, we haven't actually done anything yet.

For a test to do something, it needs to gain some information about what the test script is telling it to do. The `testHandler` class has a variable defined, called `self.xmlTestRoot`, which contains the DOM tree of the test. Specifically, it is an `ElementTree.Element` object pointing to the `<test>` element of the DOM tree. This can be navigated to extract the appropriate instructions using the interfaces provided by `ElementTree`. We advise a test developer to consult the `ElementTree` documentation for detailed instructions on the use of that library.

It should be noted that if a new test type is expected to be long-running it is useful to write the test handler in such a way that it is non-blocking, preferably in a stack-less manner using callbacks. Otherwise, the APItest browser will appear to “hang” while it is waiting for the function to exit. Examples of how to do this are shown in the `do_script()` and `do_cmd()` handlers.

D.2 The `do_cleanup_newType()` Function

The cleanup function is called as a final step during test handling. Its purpose is to provide capability to perform post-processing after a test has completed. This might include closing down a process, or deleting or archiving of temporary files. The APItest calling framework does not require anything special from this function, it works as a regular stack-based call.

D.3 The do_kill_newType() Function

The third and final function which APItest expects to find in testHandler is the kill function. This function's purpose is to shutdown the test process and exit immediately. When a timeout threshold is reached for a given test, APItest will make a call to that tests kill function. No special output is required for this function, it is invoked using a regular stack-based call.

E Output File Formats

E.1 Naming Conventions

The default mode of APItest is to save a copy of the output from a test run to disk. This output is typically saved in XML output files under a directory saved in the *oroot* directory. The format of the output directories encodes the date and time of the run.

A test run executed on September 27th, 2004 at 1:01:00 PM would be placed in a directory such as: `output/run.2004-09-27.13-01-00Z/`. Individual test results are named according to the following formula:

```
[b,t]nn.test_file.out
```

The first character is either a “b” or a “t”, corresponding to whether or not the result file is for a batch or a test file. The next chars are numbers, indicating the order in which the test or batch ran. Following the dot, we have the test file name followed by `.out`. An example batch output file might be named:

```
b12.bTest_1.out
```

A test output file might be named something like:

```
t32.testfile.out
```

Saving test output can be disabled by issuing the `-T` or `--transient` options on the command-line.

E.2 Output File for Tests

Test output format outline

```
<testResult>
  <dep/>
  :
  <output>
    <actual>
    <expect>
  </output>
  :
</testResult>
```

E.2.1 testResult element

The `testResult` element is the root level element for an APItest test result. This element contains the following attributes:

testResult Element Attributes	
filename	Filename of the test script for which this is a result.
md5sum	MD5 hash signature of the test file.
pBatchID	ID number of the batch file that called this test.
runID	ID number of the run in which this test was executed.
status	Final status of this test (PASS, FAIL)
testID	ID number of this test.
timeStart	Start time (iso8601 [14]).
timeStop	Stop time (iso8601 [14]).
timeoutFlag	Did this test timeout? “YES” or “NO”.
timeoutTime	Time limit for this test (seconds).
uid	User ID under which this test is run.
uname	User Name under which this test is run.

Table E.2. testResult Element Attributes

E.2.2 dep element

One or more of these can be contained within the `testResult` element. This element contains the status of a tests' dependencies.

- **actual** - Actual status of the parent test.
- **expect** - Expected status of the parent test.
- **parent** - Parent test file.

E.2.3 output element

The output element contains the following attributes:

- **format** - Format of the expected output? Valid values are `literal` or `regexp`.
- **matched** - Did the actual and expected output match? ("YES" or "NO").
- **name** - The name of this 'output' buffer? (i.e., `stdout` or `stderr`.)

An output element contains sub elements `<actual>CDATA</actual>` and `<expect>CDATA</expect>`, where the CDATA buffers are the actual and expected buffers specified for the test.

E.3 Output File for Batches

Test output format outline

```
<batchResult>
  <summary/>
  <child/>
  :
</batchResult>
```

E.3.1 batchResult element

The batchResult element is the root-level element for a batch file result.

- **filename** - Filename of the batch script for which this is a result.
- **md5sum** - MD5 hash signature of the test file.
- **pBatchID** - ID number of the batch file that called this batch (if any).
- **runID** - ID number of the run this batch occurred in.
- **status** - UNUSED
- **timeStart** - Start time of this batch script.

E.3.2 summary element

A batch result contains one summary element. This gives a quick summary of the contents of the file. Specifically, it gives the tally of how many tests were in this batch, how many failed, and how many passed. The attributes for the `summary` element are:

- **nFail** - Number of tests that failed.
- **nPass** - Number of tests that passed.
- **nTotal** - Total number of tests executed by this batch script.

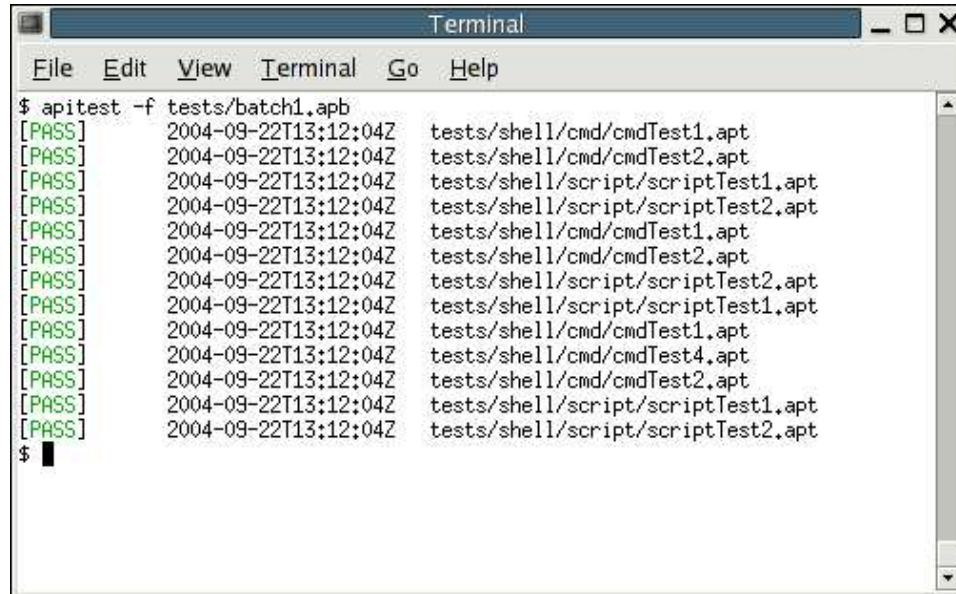
E.3.3 child element

For each test contained within this batch, there is a `child` element. This element contains data about each test that was run, its status code, and ID number. The element `<child>` contains these attributes:

- **file** - The filename of the test file run.
- **status** - Status code of this test (i.e., PASS, FAIL, FAILDEP, etc).
- **testID** - The TestID of this test. This can be used to reveal the order that the tests were run.

F Selected Screenshots

This section shows some selected screenshots of APItest in action to give a feel for what the graphical interface might look like on a user's system.

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Go", and "Help". The terminal shows the command "\$ apitest -f tests/batch1.apb" and its output. The output consists of 14 lines, each starting with "[PASS]" in green, followed by a timestamp "2004-09-22T13:12:04Z" and a file path. The file paths are: tests/shell/cmd/cmdTest1.apb, tests/shell/cmd/cmdTest2.apb, tests/shell/script/scriptTest1.apb, tests/shell/script/scriptTest2.apb, tests/shell/cmd/cmdTest1.apb, tests/shell/cmd/cmdTest2.apb, tests/shell/script/scriptTest2.apb, tests/shell/script/scriptTest1.apb, tests/shell/cmd/cmdTest1.apb, tests/shell/cmd/cmdTest4.apb, tests/shell/cmd/cmdTest2.apb, tests/shell/script/scriptTest1.apb, tests/shell/script/scriptTest2.apb, and tests/shell/script/scriptTest2.apb. The terminal ends with a prompt "\$" and a cursor.

```
$ apitest -f tests/batch1.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/cmd/cmdTest1.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/cmd/cmdTest2.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/script/scriptTest1.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/script/scriptTest2.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/cmd/cmdTest1.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/cmd/cmdTest2.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/script/scriptTest2.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/script/scriptTest1.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/cmd/cmdTest1.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/cmd/cmdTest4.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/cmd/cmdTest2.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/script/scriptTest1.apb
[PASS] 2004-09-22T13:12:04Z tests/shell/script/scriptTest2.apb
$
```

Figure F.1. Screenshot : command line

This an example terminal output from running APItest in its command-line only mode. This mode would be useful for running APItest as a scheduled run such as running a regression suite during the night.

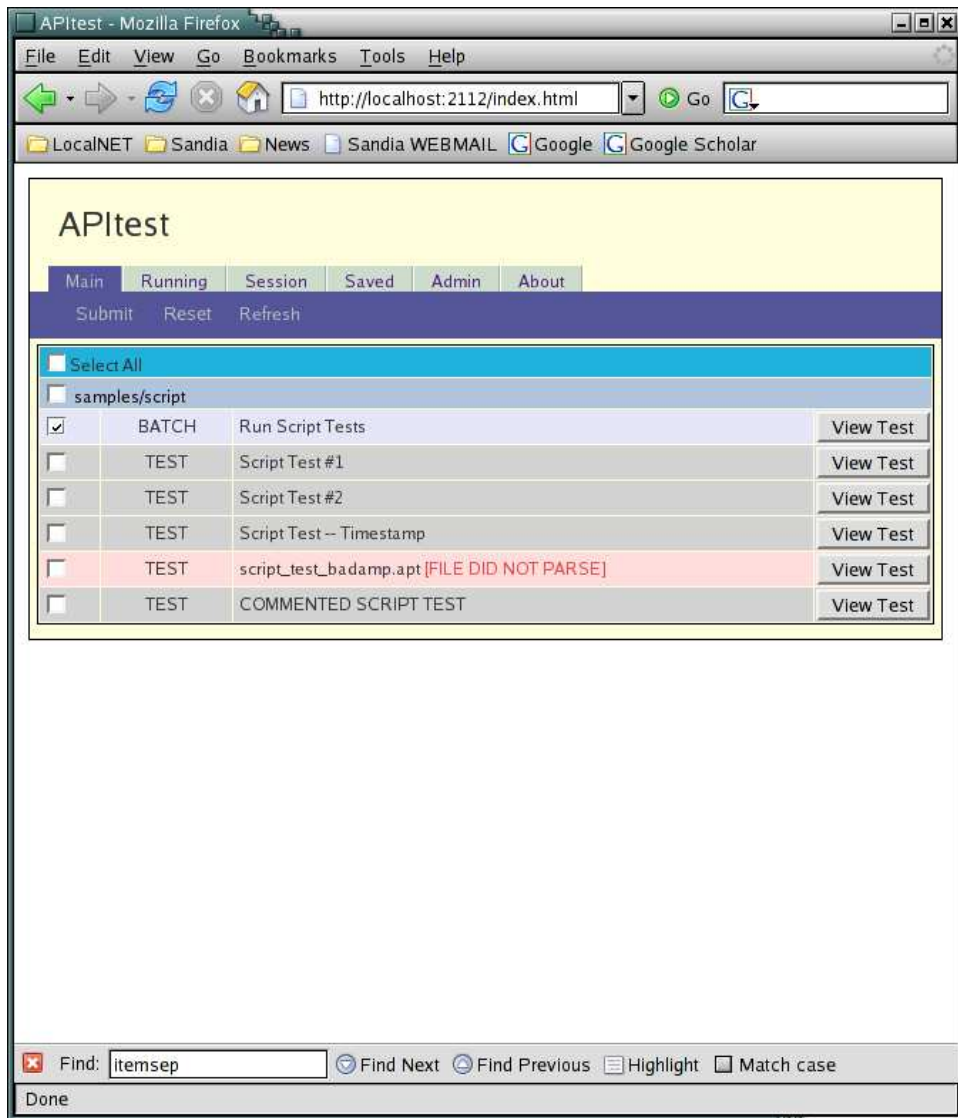


Figure F.2. Screenshot : Main

This is the main page of APItest that is shown when initially connecting to APItest. In this image we see that one test script was not properly formatted in XML; it is highlighted and marked that it did not parse correctly.

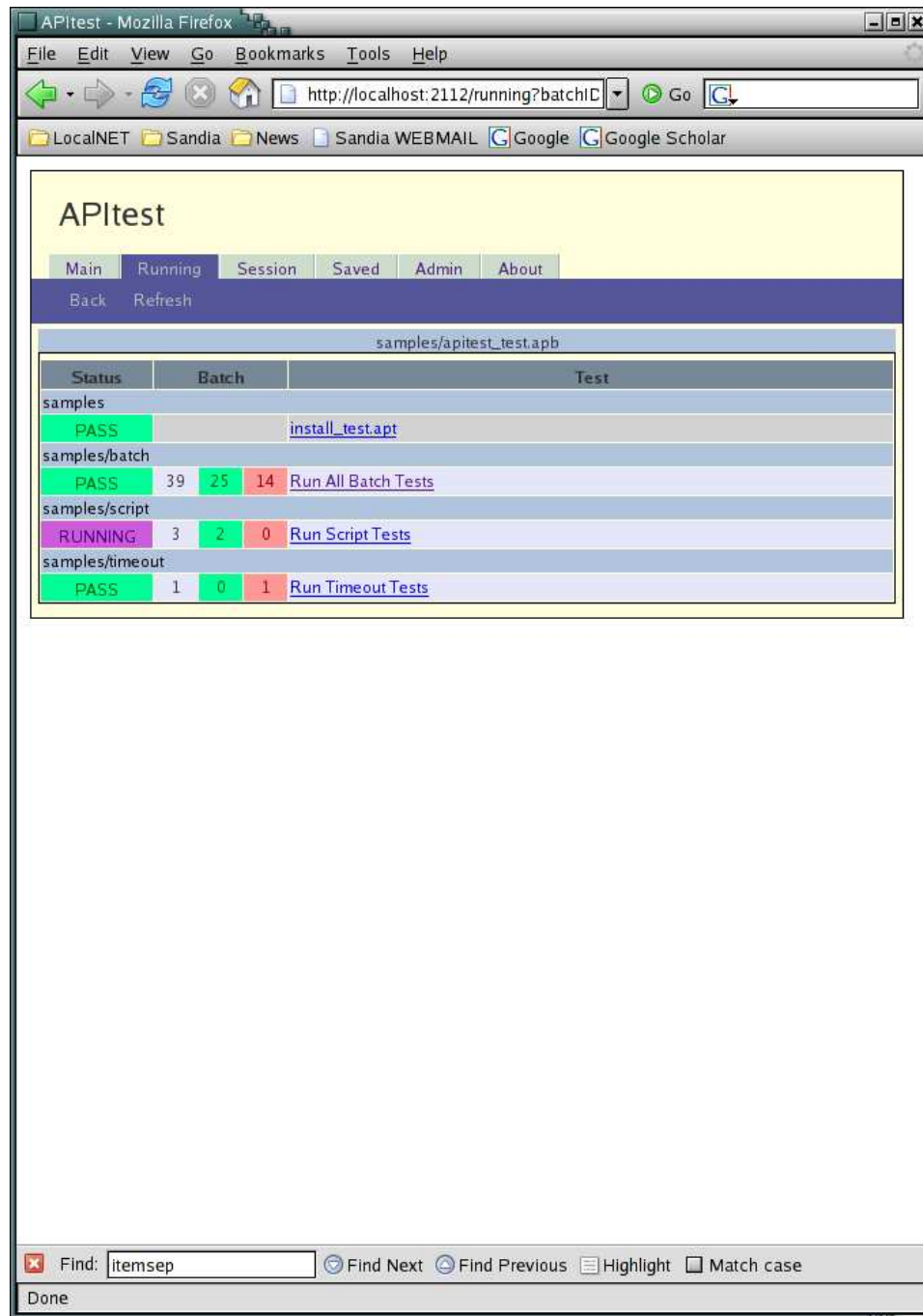


Figure F.3. Screenshot : Test Executing

During a test run tests which have run are displayed with their status. The test or batch that is currently running is indicated by **RUNNING** shown in the Status column.

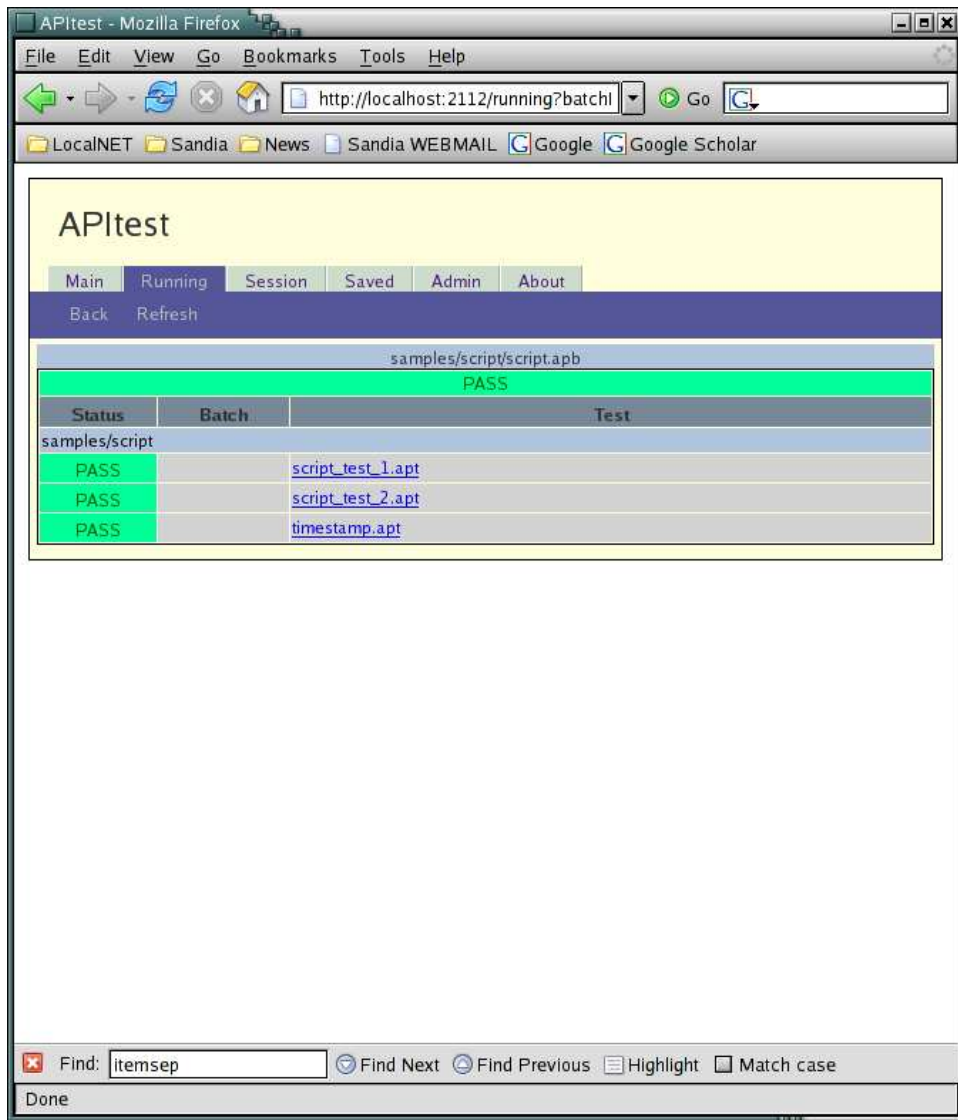


Figure F.4. Screenshot : Batch Result

Results of the list of tests run during a session. This figure shows a completed run that contained a test that timed out and some batches. The listing is ordered by directory and sorted. Batch files show their status as well as the cumulative total number of tests they executed.




Figure F.5. Screenshot : Test Result

Detailed test results for a test can be displayed. Vital statistics on the test are displayed as well as expected and actual outputs.



Figure F.6. Screenshot : Dependency Failure

Detailed results on a test that failed to meet a dependency criteria. In this case, `script_test_2` cannot run unless `script_test_1` and `cmd_test_2` pass. Unfortunately, when `cmd_test_2` fails the dependency is not met. This screenshot shows how APItest highlights the failed dependency.



```
9 # Laboratories.
10 #
11 # The redistribution of this Cplant(TM) source code is subject to the
12 # terms of the GNU Lesser General Public License
13 # (see cit/LGPL or http://www.gnu.org/licenses/lgpl.html)
14 #
15 # Cplant(TM) Copyright 1998, 1999, 2000, 2001, 2002 Sandia Corporation.
16 # Under the terms of Contract DE-AC04-94AL85000, there is a non-exclusive
17 # license for use of this work by or on behalf of the US Government.
18 # Export of this program may require a license from the United States
19 # Government.
20 #
21 #####
22 -->
23 <testDef>
24
25 <info>
26 This test should cause APITest to issue an error message because
27 the parser won't be able to know what to do with the ampersands.
28
29 Since this test file won't parse correctly, the short description
30 will not be displayed.
31 </info>
32
33 <shortDescription>Script Test -- Bad Ampersand</shortDescription>
34
35 <test type="script">
36
37 <command interpreter="/bin/csh" wdir="/tmp">
38   foreach i ('a' 'b' 'c') # &&& (bad ampersand, should be &amp;:)
=ERROR-----^
39     echo "$i"
40   end
41   echo "shell = $shell"
42 </command>
43
44 <input name="argument">-a</input>
45 <input name="argument">-b</input>
46
47 <output name="stdout" format="regex">x\nb\nc</output>
48 <output name="stderr" format="literal">a</output>
49 <output name="status" format="literal">0</output>
50
51 </test>
52
53 </testDef>
54
55 <!-- EOF -->
```

Figure F.7. Screenshot : XML Error Detection

Example showing an XML input file that contained an error. APITest prints out the text of the file and marks the location that the parser encountered the error.

G Troubleshooting

- **Menus don't display correctly in my browser.**

This is because the menu is done using stylesheets, and some browsers might have problems with it. I have tested APItest in Safari, Firefox, Mozilla, and on MSIE under Windows XP.

APItest can be run in a compatibility mode to turn off the CSS menus. Running with the following command line may work:

```
$apitest httpd --nocss
```

- **I have a script test that won't parse. APItest complains about my use of & and < symbols.**

If your script has a & symbol in it, it will confuse the parser because the & character is a special character for XML. This can be easily resolved by replacing every occurrence of & with &.

The < symbol will confuse the parser because it is also a special token. When the parser finds a <, it believes it is the start of a new XML element. We can resolve this by replacing the < symbol with <.

- **I made APItest run a script that puts itself into the background (i.e., daemonizes itself), and APItest seems to hang. It doesn't make any further progress on its tests.**

This happens because of the nature of some of the internals of APItest. It does not currently support running multiple tests in parallel. Each test must finish before the next test will run.

Even though the script is daemonized, it still is keeping pipes open to stdout and stderr for output. The script handler sees this and realizes the script is still running.

APItest can be tricked into thinking that the script has completed by having the script close its pipes to stdout and stderr. This should allow APItest to continue on with subsequent tests. Don't forget to have the daemon terminated after the tests are finished.

H List of Installed Files

Installing the RPM distribution installs the following files. The site-packages directory may be different if APItest is installed via `python setup.py install`, depending upon your environment.

Site Packages

```
/usr/lib/python2.3/site-packages
/usr/lib/python2.3/site-packages/libapitest
/usr/lib/python2.3/site-packages/libapitest/__init__.py
/usr/lib/python2.3/site-packages/libapitest/digraph.py
/usr/lib/python2.3/site-packages/libapitest/digraph.pyc
/usr/lib/python2.3/site-packages/libapitest/htmltools.py
/usr/lib/python2.3/site-packages/libapitest/htmltools.pyc
/usr/lib/python2.3/site-packages/libapitest/httpHandler.css.py
/usr/lib/python2.3/site-packages/libapitest/httpHandler.css.pyc
/usr/lib/python2.3/site-packages/libapitest/imageHandler.py
/usr/lib/python2.3/site-packages/libapitest/imageHandler.pyc
/usr/lib/python2.3/site-packages/libapitest/jobManager.py
/usr/lib/python2.3/site-packages/libapitest/jobManager.pyc
/usr/lib/python2.3/site-packages/libapitest/libapitest.py
/usr/lib/python2.3/site-packages/libapitest/libapitest.pyc
/usr/lib/python2.3/site-packages/libapitest/libdebug.py
/usr/lib/python2.3/site-packages/libapitest/libdebug.pyc
/usr/lib/python2.3/site-packages/libapitest/stylesheets.py
/usr/lib/python2.3/site-packages/libapitest/stylesheets.pyc
/usr/lib/python2.3/site-packages/libapitest/systools.py
/usr/lib/python2.3/site-packages/libapitest/systools.pyc
/usr/lib/python2.3/site-packages/libapitest/testHandler.py
/usr/lib/python2.3/site-packages/libapitest/testHandler.pyc
/usr/lib/python2.3/site-packages/libapitest/twistedTools.py
/usr/lib/python2.3/site-packages/libapitest/twistedTools.pyc
```

APItest Executable

```
/usr/local/apitest/apitest
```

Samples

```
/usr/local/apitest/samples/apitest_test.apb
/usr/local/apitest/samples/batch
```

/usr/local/apitest/samples/batch/bTest_1.apb
/usr/local/apitest/samples/batch/bTest_2.apb
/usr/local/apitest/samples/batch/bTest_3.apb
/usr/local/apitest/samples/batch/bTest_4.apb
/usr/local/apitest/samples/batch/bTest_5.apb
/usr/local/apitest/samples/batch/bTest_6.apb
/usr/local/apitest/samples/batch/bTest_7.apb
/usr/local/apitest/samples/batch/bTest_8.apb
/usr/local/apitest/samples/batch/bTest_conditional.apb
/usr/local/apitest/samples/batch/bTest_conditional_FAIL.apb
/usr/local/apitest/samples/batch/bTest_conditional_PASS.apb
/usr/local/apitest/samples/batch/bTest_faildep.apb
/usr/local/apitest/samples/batch/bTest_seq.apb
/usr/local/apitest/samples/batch/bTest_subTestCheck.apb
/usr/local/apitest/samples/batch/batch.apb
/usr/local/apitest/samples/cmd
/usr/local/apitest/samples/cmd/cmd.apb
/usr/local/apitest/samples/cmd/cmd_filediff.apt
/usr/local/apitest/samples/cmd/cmd_nomatch.apt
/usr/local/apitest/samples/cmd/cmd_notfound_1.apt
/usr/local/apitest/samples/cmd/cmd_notfound_2.apt
/usr/local/apitest/samples/cmd/cmd_test_1.apt
/usr/local/apitest/samples/cmd/cmd_test_2.apt
/usr/local/apitest/samples/cmd/cmd_test_3.apt
/usr/local/apitest/samples/cmd/cmd_test_4.apt
/usr/local/apitest/samples/daemon
/usr/local/apitest/samples/daemon/daemonize.apt
/usr/local/apitest/samples/envvar
/usr/local/apitest/samples/envvar/envvar.apb
/usr/local/apitest/samples/envvar/envvar_cmd.apt
/usr/local/apitest/samples/envvar/envvar_cmd_suid.apt
/usr/local/apitest/samples/envvar/envvar_script.apt
/usr/local/apitest/samples/envvar/envvar_script_suid.apt
/usr/local/apitest/samples/install_test.apt
/usr/local/apitest/samples/io
/usr/local/apitest/samples/io/io.apb
/usr/local/apitest/samples/io/io_large_matchall.apt
/usr/local/apitest/samples/io/io_large_miss_stderr.apt
/usr/local/apitest/samples/io/io_large_miss_stdout.apt
/usr/local/apitest/samples/scidac-sss

```
/usr/local/apitest/samples/scidac_sss/sd
/usr/local/apitest/samples/scidac_sss/sd/sss_01.apb
/usr/local/apitest/samples/scidac_sss/sd/sss_02.apb
/usr/local/apitest/samples/scidac_sss/sd/sss_sd_inittest.apb
/usr/local/apitest/samples/scidac_sss/sd/sss_sd_inittest_cleanup.apb
/usr/local/apitest/samples/scidac_sss/sd/sss_sd_inittest_prep.apb
/usr/local/apitest/samples/scidac_sss/sd/sss_sd_remove_emng.apb
/usr/local/apitest/samples/scidac_sss/sd/sss_sdstat_sdoft.apb
/usr/local/apitest/samples/scidac_sss/sd/sss_sdstat_sdon.apb
/usr/local/apitest/samples/scidac_sss/sd/sss_start.apb
/usr/local/apitest/samples/scidac_sss/sd/sss_stop.apb
/usr/local/apitest/samples/script
/usr/local/apitest/samples/script/script.apb
/usr/local/apitest/samples/script/script_foo.apb
/usr/local/apitest/samples/script/script_test_1.apb
/usr/local/apitest/samples/script/script_test_2.apb
/usr/local/apitest/samples/script/script_test_badamp.apb
/usr/local/apitest/samples/script/timestamp.apb
/usr/local/apitest/samples/suid
/usr/local/apitest/samples/suid/suid_cmd_uid.apb
/usr/local/apitest/samples/suid/suid_cmd_uname.apb
/usr/local/apitest/samples/suid/suid_cmd_uname_notfound.apb
/usr/local/apitest/samples/suid/suid_cmd_uname_wdir_1.apb
/usr/local/apitest/samples/suid/suid_cmd_uname_wdir_2.apb
/usr/local/apitest/samples/suid/suid_script_uid.apb
/usr/local/apitest/samples/suid/suid_script_uname.apb
/usr/local/apitest/samples/suid/suid_script_uname_notfound.apb
/usr/local/apitest/samples/suid/suid_script_uname_status.apb
/usr/local/apitest/samples/timeout
/usr/local/apitest/samples/timeout/timeout.apb
/usr/local/apitest/samples/timeout/timeout.apb
```

DISTRIBUTION:

1	Thomas Naughton (ORNL)	1	MS 1110	
			Ron Oldfield, 9223	
1	Al Geist (ORNL)			
1	Scott Jackson (PNNL)	1	MS 0817	
			James Ang, 9224	
1	Rusty Lusk (ANL)			
1	Paul Hargrove (LBL)	1	MS 0823	
			John Zepper, 9320	
1	Craig Steffan (UIUC)	1	MS 0807	
			John Noe, 9328	
1	Brett Bode (AMES)			
1	MS 0321	1	MS 0806	
	Bill Camp, 9200		Leonard Stans, 9336	
1	MS 1110	1	MS 0139	
	Steve Plimpton, 9212		Art Hale, 9900	
1	MS 1110	1	MS 9151	
	William Hart, 9215		Jim Handrock, 8960	
1	MS 1110	1	MS 9158	
	Neil Pundit, 9223		Mitch Sukalski, 8961	
1	MS 0817	1	MS 9158	
	Doug Doerfler, 9220		Robert Armstrong, 8961	
1	MS 0140	1	MS 9018	
	Robert Leland, 9220		Central Technical Files,	
			8940-2	
10	MS 1110	2	MS 0899	
	William McLendon, 9223		Technical Library, 4916	