Test Environment Toolkit

Architectural, Functional, and Interface Specification

Project Phoenix
Revision: 4.5 (Date: 92/02/18)

1. Introduction

The Test Environment Toolkit (TET) Project started in September of 1989. At that time, X/Open, the Open Software Foundation, and UNIX International entered into an announced agreement to produce a specification for a test environment. These organisations also agreed to develop and make freely available an implementation written to that specification. Additionally, the three organisations committed to producing tests suites for execution within that environment.

In the process of developing a specification, the project invited interested members of the test software development community to discuss their requirements for a test driver. These requirements, which can be seen in Appendix B, are cited throughout this specification using the notation *[Requirement n.n]*. It was the belief of the Project that, through careful study of these requirements, a reasonably comprehensive and effective test driver could be specified. Having achieved this, the Project now expects that a substantial portion of the test development community will begin using this environment for the development of conformance testing software.

The primary goal of this project, as is reflected in the requirements, was to define a specification which would guarantee portability of the TET architecture across a wide variety of systems. While it is essential that TET operate in UNIX® like environments, every caution has been taken to ensure that TET is implementable on any platform which can satisfy the system requirements (§3.4).

A ramification of working toward this goal is that much of this specification is not tied very tightly to any implementation of TET. While this is intentional, a short guide to the document may make it easier to understand.

Section 3 provides an architectural overview of TET. This includes the system requirements and test suite requirements. It also defines terminology that is used throughout the document. Section 4 is a system-independent functional specification. This describes the functionality provided by TET in such a way that it can be bound to in a number of different languages on any system satisfying the requirements in section 3. Sections 5 and 6 are where all of the TET language bindings appear. These are bindings for specific TET functions in specific environments. The bindings provided in this specification pertain to POSIX.1 systems with either ANSI C or common C support, and XPG3 shell. Section 7 defines the formats for TET portable input and output data files.

® **UNIX is a registered trademark of UNIX System Laboratories in the United States and other countries.**

2. Scope

This specification describes a test controller, a test suite installation tool, a test report generator, and programmatic interfaces to be used in the development of test cases. The Project has tried to allow for future extensibility and flexibility of the Toolkit so that test developers can work within the structures of the driver and still produce useful, effective tests.

What the project has not addressed is the formulation, content, or scope of test cases. A test case can be arbitrarily complex or simple and still function under the Toolkit. The structure of a test case can be based on assertions, functions, modules, or any other appropriate abstraction. The test suite developer must choose the most useful structure for a given test suite. This choice will often be based on requirements of various conformance specifications. Because of the overriding concern for providing flexibility for the test developer, the Toolkit imposes a minimum amount of structure on test cases. *[Requirement 3.8]*

3. Architecture

The architecture of the Test Environment Toolkit (TET) provides simplicity and portability, which will lead to transportability of suites. The structure of TET is intuitive, and the required tools have consistent structure and interfaces. *[Requirement 1.3]* The architecture can be described in terms of four objects:

- The Terminology, which loosely defines the elements of the TET conceptual model.
- The Conceptual Model, which shows how elements in the TET structure link together.
- The System Requirements, which define minimum system facilities necessary to support an implementation of the TET functionality.
- The Test Suite structural requirements, which show how a Test Suite must be constructed to function within a conforming TET implementation.

The remainder of this section describes these four objects.

3.1 Terminology

The following terms are used throughout this document.

Application Programming Interface
: An application programming interface (API) is the set of software interfaces between an application and the system. In the case of TET, the API provides a Test Case Manager (TCM), which provides an interface between the test case and the test case controller, and an API library, which provides facilities for use by test cases.

Buildfile
: A buildfile is a set of test suite defined instructions to the buildtool for use in constructing executable versions of test cases.

Buildtool
: The test case controller uses a test suite defined buildtool to control the construction of executable versions of test cases. This buildtool uses a test suite defined buildfile as instructions for building each test case. In order for configuration variables to be available to the buildtool it must use the TET API.

Build Fail Tool
: A test suite defined utility which is executed by the test case controller when the buildtool indicates that a test case build has failed.

Cleanfile
: A cleanfile is a set of test suite defined instructions to the cleantool for use in removing executable versions of test cases and associated files.

Cleantool
: The test case controller uses a test suite defined cleantool to control the removal of executable versions of test cases and associated files. This cleantool uses a test

suite defined cleanfile as instructions for cleaning up each test case. In order for configuration variables to be available to the cleantool it must use the TET API.

Communication Variables

The TCC uses communication variables to provide information to the buildtool, cleantool, and test cases during execution. On some implementations communication variables may share namespace with information outside of TET. In these cases implementors are responsible for ensuring that users and test suite authors are informed of the potential for collision.

Configuration Variables

The TCC and API use configuration variables to change operation behaviour. Configuration variables are set via configuration variable files (§7.1.1) and via the TCC user interface. Configuration variables should exist within a namespace private to an instance of TET. TET-defined configuration variable names all begin with the sequence "TET". Test suite authors are cautioned to use obvious and consistent naming conventions to avoid potential conflicts with other objects in the namespace.

Execfile

An execfile is a set of test suite defined instructions for use in executing test cases under control of an exectool.

Exectool

A tool used for executing test cases under special control, e.g. a debugger. Normally no exectool is specified, which means test cases are executed directly.

Execution Results

A file into which test cases, via the API, place information regarding their execution. This information is later incorporated into the journal by the TCC.

Invocable Component

An invocable component is the smallest unit that the TCC can execute individually. Invocable components are made up of one or more test purposes.

Journal

A journal is the file into which test results and tracking data are deposited by the TCC.

Result Code

A result code is the determination made by a test case about a test purpose. TET supports the result codes defined by IEEE Std 1003.3-1991, as well as additional, test suite defined results. Results are recorded via the API, and can be analysed by the report generator and appropriate treatment filters.

Test Case

A test case is the software which conducts tests. The scope of the term "test" is broad. It may range from a single test purpose for a single function being tested, all the way to a complete suite of conformance tests for a specification. Test cases are made up of one or more invocable components.

Test Case Controller
: A test case controller (TCC) is the tool that provides structure and control for test cases. The TCC builds test cases when invoked in build mode, removes them in clean mode, and executes the invocable components within test cases when in execute mode. The tool handles such functions as sequencing of test cases, parameter passing, and journal file constructs.

Test Case Manager
: The test case manager is a component of the API. This component interfaces between the test case controller and the test suite defined parts of the test case. It provides support for the automatic sequencing of test purposes and invocable components, as well as insulation from the testing environment.

Test Purpose
: A test purpose is the software which tests that statement which is the smallest level of granularity of a test specification. A test purpose always leads to a single result. In the case of a POSIX.3 conforming test suite, for example, test purposes would correspond to assertions.

Test Scenario
: A test scenario is a sequence of one or more invocable components associated with a single user-exposed name. There may be multiple test scenarios per scenario file. Test scenarios are used by the TCC to translate user requests into actions. *[Requirement 3.1]*

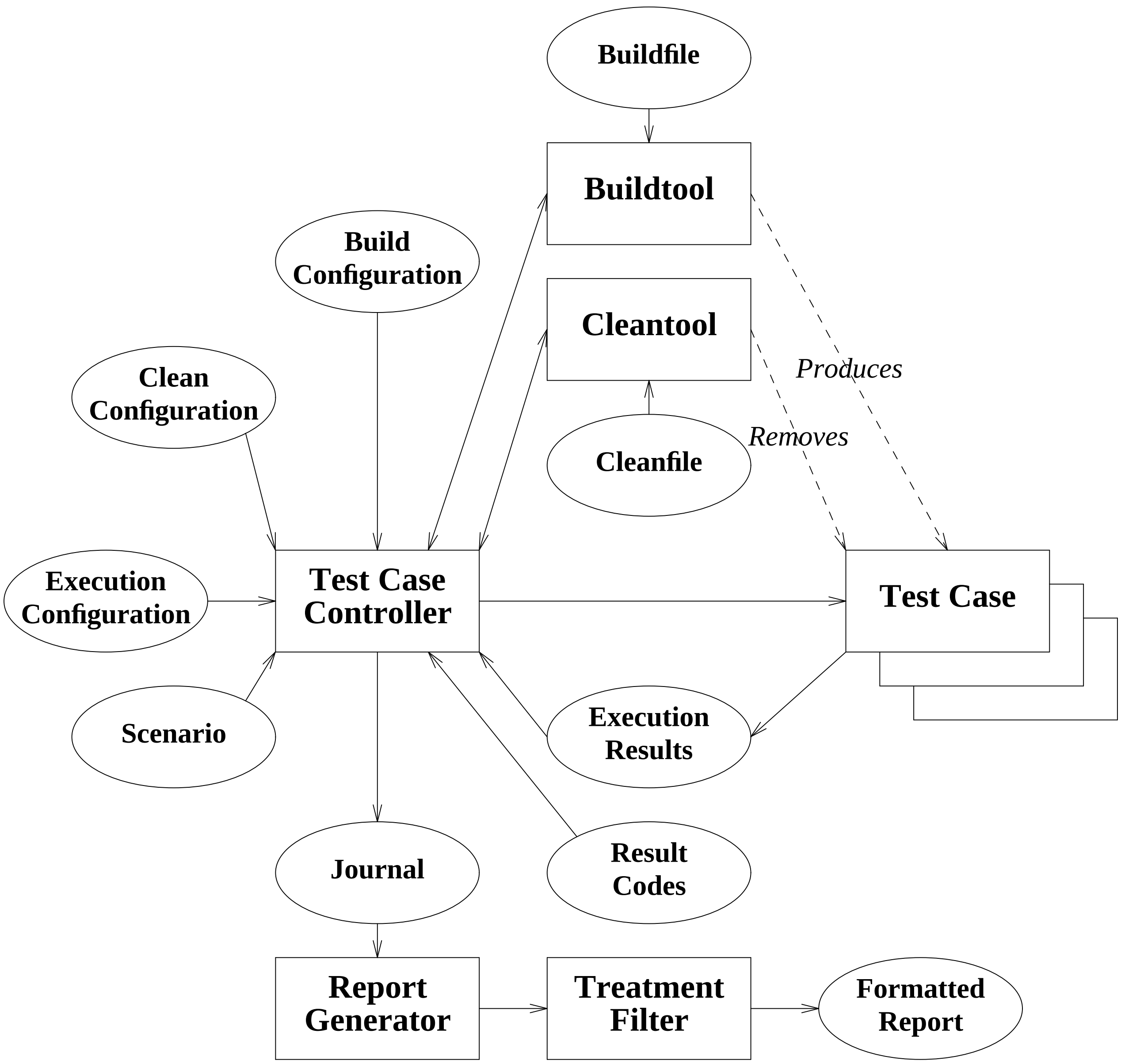
Test Suite
: A test suite is a set of test cases and other required and optional files. A single test case is a minimal example of a test suite.

3.2 Conceptual Model

The TET architecture consists of a number of functions and data files. The functions do not necessarily relate to specific executables in the implementation. Neither do the data files necessarily relate to specific files. These are considered to be implementation details which are beyond the scope of the architecture description. This description is intended to be platform-independent within the resource requirements detailed in section 3.4. *[Requirements 1.1 & 1.6]*

The conceptual model chosen consists of six functions and ten data files that together describe TET.

TET Conceptual Model

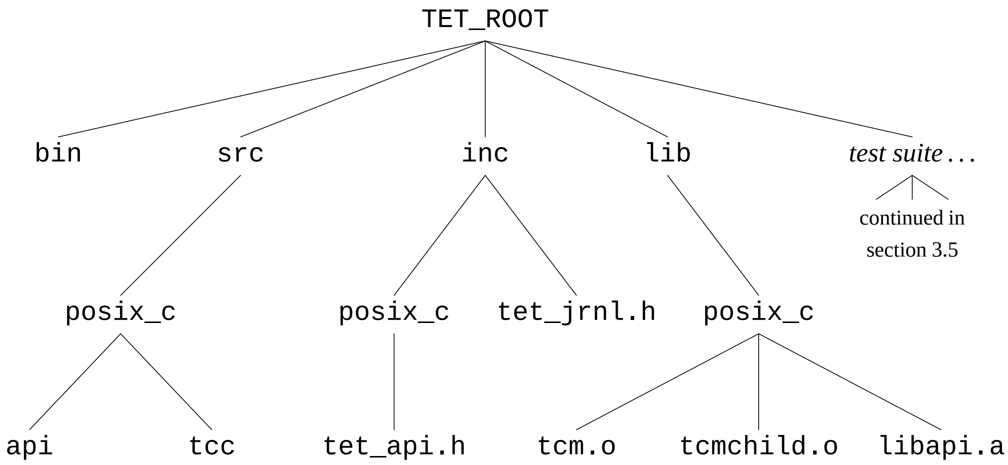


Each of the functions and data files indicated above is described in detail in the following sections.

3.3 Structural Description

The following diagram shows how the various components of TET and test suites relate to one another:

TET Directory Structure



3.4 System Requirements

In order for a system to support the toolkit architecture, the following facilities must be available.

Execution of Discrete Components

> Systems must allow applications to execute other applications. These applications must have a hierarchical relationship such that an application which launches another can have some control over the other's execution. Further, there must be a mechanism for uniquely identifying each application concurrently executing in the system.

Files and Directories of Files

> TET software requires a system with the ability to read and write data resources ("files"). These files must be structured in such a way that directories of files can be separated from other directories. These directories must also be combinable into other directories, with the memberships of the various groupings maintained. Files within each directory are independently accessible.

Resource Protection Mechanism

> Support to ensure that a specified resource is protected from access by other processes executing concurrently.

Atomic File Writes

> The system must permit atomic writes of at least 512 bytes to files.

Options/Operands

> The system must provide a method of providing applications with user-specified parameters. The order of user-specified parameters must be maintained.

Date/Time Facility

> The system must provide a way for an application to determine the date and time of day. While there is no absolute requirement on granularity of such a timer, one

second granularity is advisable.

Completion Status

Upon termination of an application, the system must provide a way of delivering a status of that termination to the application which controls the terminated application. Further, the system must have a concept of "normal" completion. TET relies upon "normal" and "abnormal" execution completion in some modes (§4.1).

3.4.1 Future Directions

Proposed future additions to TET will require systems to provide the following facilities.

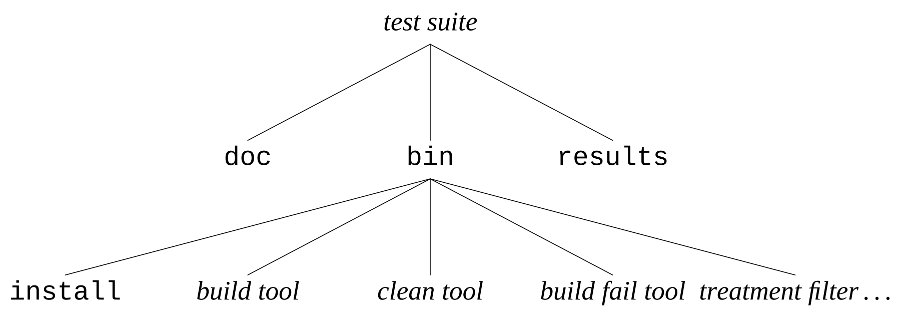
Internationalisation

A future revision of the specification will require that TET implementations support the concept of internationalisation. This includes the use of Message Libraries, Locale-dependent behaviour, and wide-character based string manipulation. Systems will need to provide the relevant facilities. *[Requirement 2.3]*

3.5 Test Suite Structure

The Toolkit attempts to impose minimum structural requirements on test suites. However, in order to allow the maximum flexibility of those suites, some specific files and utilities must be present. A rough representation of this structure is:

Test Suite Directory Structure



3.5.1 Required Files

Execution Results Files

As tests are executed, their results are accumulated in execution results files. The TCC processes those files upon completion of test case execution. Execution results must be placed in a file called `tet_xres` in the same directory as test case execution took place. The format of this file is defined in Journal Files (§7.2.1).

Result Code Definitions

The result code definitions file contains the test suite specific definitions for possible test purpose results. This file is used by the API and test case controller when interpreting test purpose results. The format of this file is defined in Result Code Definitions (§7.1.3).

Configuration Variable Settings

TET has several configuration variables which allow test suites to customise the way in which the test case controller operates. Test suites may define additional configuration variables as needed. Files containing the test suite settings for these variables are used by the API and test case controller when building, executing and cleaning up test cases. The format of these files is defined in Configuration Variable Files (§7.1.1).

Test Scenario Definitions

Each test suite must provide a test Scenario definition file. This file (by default named `tet_scen`) contains sequences of invocable components and a related, single name that can be used by the TCC and Toolkit users when executing the test suite. As a minimum a scenario named **all** must be provided. This scenario should execute all invocable components in all test cases in the test suite. The format of a scenario file is defined in Scenario Files (§7.1.2).

Tools Directory

Each test suite must have a directory in its top level called `bin`. This directory contains tools that are used by the test suite and the TCC.

Results Directory

Each test suite has within it a directory called `results`. The TCC creates a unique subdirectory in this directory on each run, into which it places the journal file and a hierarchy of files requested to be saved by the user.

Documentation Directory

Each test suite is required to provide a directory in its top level called `doc`. This directory contains appropriate documentation for the test suite.

3.5.2 Required Utilities

Since TET cannot anticipate the internal structure of test suites, it uses test-suite-supplied utilities to perform some tasks. In particular, TET requires that the test suite provide utilities to assist in installing the test suite.

3.5.3 Optional Utilities

Test Suites may provide some additional tools for TET users.

Build Tool

This utility is executed by the TCC to build each test case. The build tool is invoked in the source directory of the test case. If the communication parameter TET_EXECUTE is set, it is the responsibility of the build tool to install the relevant files under the specified alternate execution directory.

Clean Tool

This utility is executed by the TCC to clean each test case. The clean tool is invoked in the source directory of the test case. If the communication parameter TET_EXECUTE is set, it is the responsibility of the clean tool to remove the relevant files from under the specified alternate execution directory.

Build Fail Tool

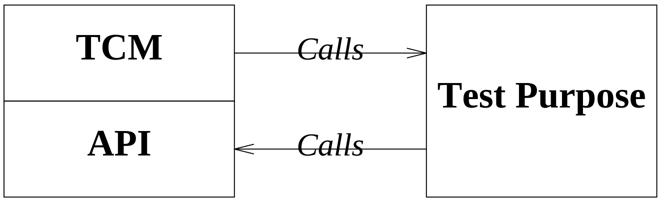
This utility is optionally activated by the TCC when the buildtool indicates that a test case build failed. The build fail tool is invoked in the directory where the build took place with the name of the test case as an argument. One possible use for such a utility would be to provide a skeleton test case which merely indicated that the test case was not built successfully when executed.

Treatment Filters

Test suite authors may wish to provide treatment filters that produce reports in specific formats. The format of the journal file is defined in Journal Files (§7.2.1).

3.6 Test Case Structure

Test cases are constructed by grouping together routines which test specific system features. These routines take advantage of support functions provided by the TET API, and are called by the TET TCM. This relationship is best illustrated through the following picture:



This diagram shows that the TCM calls test purposes, which in turn make calls into the API. The API performs functions like fetching configuration variable settings and writing messages into the journal. By concentrating this functionality in the API, and by allowing the TCM to control test sequencing, TET offers the test suite author strong support for developing test suites.

Test purposes within a test case may be grouped together into invocable components. In doing so, a test suite author can ensure that a set of test purposes is always executed together and in the correct sequence. There is no limit to the number of test purposes that can be in an invocable component, and there is no limit to the number of invocable components that can be within a single test case.

The advantage of this structure is that it allows the test suite author a great deal of flexibility, while still presenting a consistent interface for the test suite user.

Details about the functionality provided by the TCM and API can be found in the Application Programming Interface functional specification (§4.3). Specific information about constructing test cases in various programming languages and environments is available in Application Programming Interfaces (§6).

4. Functional Specifications

The functional specification provides system-independent descriptions of the technology required by the Toolkit's architecture. The descriptions include sufficient detail to implement a conforming environment. The functionality described in this section must be fully supported by any TET implementation. Interface specifications which support this functionality in specific environments are provided in Sections 5 and 6.

4.1 Test Case Controller

This section specifies the functionality of the Test Case Controller (TCC). The TCC determines user and suite specific configuration options and facilitates user control of test sessions. This control includes the building and execution of test cases.

In addition to the control of test sessions, the TCC also includes functionality to support internal mechanisms essential to the operation of the Toolkit environment. These include managing interaction with the test case, the processing of results and the removal of temporary files.

Note that while the TET architecture is designed to support multiple test suites simultaneously, the TCC only permits execution of test cases from a single test suite per TCC run.

4.1.1 Modes of Operation

The TCC provides these three modes of operation:

- Build - translate source test cases into executables
- Execute - load and execute test cases
- Clean - remove files created by build mode

The TCC operates in any combination of build, execute or clean modes. The behaviour of these modes may be modified by user direction or by configuration options.

Regardless of the mode selected, the TCC performs the following actions during start-up:

1. If a specific journal is named by the user, verifies that it does not exist.
2. Verifies that the test suite selected by the user is available.
3. Identifies the selected scenario file, and ensures that the scenario selected by the user is present.
4. Identifies configuration variable settings, and integrates custom user settings for those configuration variables.
5. Ensures that the results directory exists.
6. Tells the user the name of the journal file being used.
7. Writes a start-up message into the journal.

8. Delivers details of option and configuration variable settings for all modes enabled to the journal file.

After start-up the TCC processes the selected scenario, performing operations on each test case in the order they are encountered. *[Requirement 3.10]* The following sections describe TCC actions in each mode when test cases are encountered.

4.1.1.1 Build Mode

When the test suite is provided in source form, the TCC is responsible for building executable versions of test cases. There is no requirement that test suites be provided in source form. Therefore use of build functionality is at the discretion of the test suite provider and user (i.e., binary distribution of tests is supported, assuming the binary is suitable for the system under test). *[Requirement 2.4]*

For each test case referenced in the specified test scenario, the TCC does the following:

1. Delivers a build start message to the journal.
2. Obtains exclusive locks in the source and execution directories of the test case.
3. Executes buildtool in the source directory of the test case. If output capture mode is enabled, then the arguments passed to buildtool are TET_BUILD_FILE and the name of the test case, otherwise no arguments are passed.
4. Transfers buildtool output and error messages to the journal file, if output capture mode is enabled. If output capture mode is disabled, re-orders and copies the contents of the results file to the journal.
5. Removes the lock set in step 2.
6. Delivers a build end message to the journal.

If the user has specified an alternate execution directory, TCC provides that information to the buildtool via a communication variable (§4.1.4). Buildtool and buildfile authors must ensure that appropriate files are transferred to the alternate execution directory during the build operation.

In build mode the user is able to:

- Set the scenario file name.
- Set the build configuration file name.
- Set the root directory for the hierarchy of test case executables.
- Set the name of the journal file.
- Set the timeout limit (in seconds) per buildtool execution.
- Select the re-run or resume options (see section 4.1.2).
- Set configuration variable values.

4.1.1.2 Execute Mode

In this mode, the TCC executes test cases in the specified scenario. For each test case referenced, the TCC does the following:

1. Delivers a test case start message to the journal. *[Requirement 3.9]*
2. Obtains a shared or exclusive lock in the execution directory of the test case. If TET_EXEC_IN_PLACE is false, the lock is shared, otherwise exclusive.
3. If TET_EXEC_IN_PLACE is false, creates a temporary directory in which executions can be safely performed. *[Requirement 3.7]*
4. If TET_EXEC_IN_PLACE is false, recursively copies the files in the test case execution directory to the temporary directory.
5. Removes the lock set in step 2, if TET_EXEC_IN_PLACE is false.
6. If TET_EXEC_TOOL is not defined, executes the test case in the test case execution directory if TET_EXEC_IN_PLACE is true, otherwise in the temporary directory, with arguments of the appropriate invocable components. If TET_EXEC_TOOL is defined, executes the exectool in the test case execution directory if TET_EXEC_IN_PLACE is true, otherwise in the temporary directory, with arguments of TET_EXEC_FILE, the test case name and the appropriate invocable components. Behaviour of test cases written to the TET API is detailed in Application Programming Interfaces (§4.3).
7. Recursively transfers files matching TET_SAVE_FILES to the corresponding place in the saved files directory hierarchy.
8. If output capture mode is disabled, re-orders and copies the contents of the execution results file to the journal. If any test purpose has not generated a result, the TCC adds a result message of `NORESULT`. If output capture mode is enabled, the TCC adds appropriate lines to the journal based on test case output and exit status (§4.1.3).
9. Removes the lock set in step 2, if TET_EXEC_IN_PLACE is true.
10. Removes the temporary directory, if TET_EXEC_IN_PLACE is false,
11. Delivers a test case end message to the journal.

In execute mode the user may:

- Set the scenario file name.
- Set the execution configuration file name.
- Set the alternate execution directory.
- Set the root directory for the saved files hierarchy.
- Set the name of the journal file.

- Set the timeout limit (in seconds) per test case execution.
- Set the list of file names which are to be copied to the saved files hierarchy after each test case has completed.
- Select the re-run or resume options (see section 4.1.2).
- Set configuration variable values.

4.1.1.3 Clean Mode

Users may request that the TCC perform operations on test scenarios to remove unwanted files following test sessions. Clean mode will never affect the results of previous test runs.

For each test case referenced, the TCC does the following:

1. Delivers a clean start message to the journal.
2. Obtains exclusive locks in the source and execution directories of the test case.
3. Executes cleantool in the source directory of the test case. If output capture mode is enabled, then the arguments passed to cleantool are TET_CLEAN_FILE and the name of the test case, otherwise no arguments are passed.
4. Transfers cleantool output and error messages to the journal file, if output capture mode is enabled. If output capture mode is disabled, re-orders and copies the contents of the results file to the journal.
5. Removes the lock set in step 2.
6. Delivers a clean end message to the journal.

In clean mode the user may:

- Set the scenario file name.
- Set the clean configuration file name.
- Set the root directory for the hierarchy of test case executables.
- Set the timeout limit (in seconds) per clean.
- Select the re-run or resume options (see section 4.1.2).
- Set configuration variable values.

4.1.2 Major Options

4.1.2.1 Resume Option

When invoked with the resume option, the TCC analyses the old journal file, searching for user-specified result codes and build and clean status codes. When one of the specified result codes or status codes is found in the journal file, the TCC builds, executes and/or cleans that and all subsequent invocable components in the scenario.

Note that this option requires the existence of the scenario file which was used to generate the old journal file. *[Requirement 3.3]*

The user specifies the search criteria as a list of result code names and other code letters which the TCC processes as follows:

List element	**Journal lines matched** (see 7.2.1)
result_code_name	Test Purpose Result lines giving execution results (not build or clean results) with the equivalent result code number.
b	Build End lines in which the completion status indicates a failed build; and, Test Purpose Result lines giving build results with any non-zero result code number.
e	Test Case End lines in which the completion status indicates the test case did not run to completion; and, Test Purpose Result lines giving execution results with any non-zero result code number.
c	Clean End lines in which the completion status indicates a failed clean; and, Test Purpose Result lines giving clean results with any non-zero result code number.

4.1.2.2 Re-run Option

This option is similar to the resume option, with the exception that **only** invocable components which meet the user-specified search criteria will be processed. *[Requirement 3.6]* Where the search finds test cases which did not run to completion, all the invocable components of that test case that are listed in the scenario file and for which there is no IC End line in the journal are processed.

4.1.3 Configuration Variables

Configuration variables allow the modification of defaults for the TCC and test suites. The mechanism for supporting this functionality is a configuration file per test suite per mode. *[Requirement 2.2]*

For each mode the configuration information is obtained from a configuration file. This file is located by the TCC using the following algorithm: *[Requirement 4.3]*

1. If the user named a specific configuration variable file at invocation time, that file is used;
2. otherwise, if the user-named alternate execution directory has a configuration variable file in it, that file is used;
3. otherwise, if there is a configuration variable file in the test suite root directory, that file is used.

If the user set specific variables at TCC invocation time, those settings override the ones in the configuration variable file.

Configuration data is made available to the test case via the API. The mechanism for providing this information is implementation defined.

Test suites may add their own configuration variables, although variable names beginning with the sequence "TET_" are reserved for future extensions of the Toolkit. There are no default values for test suite defined variables. Test suite authors must ensure that in the absence of necessary configuration information their tests behave appropriately.

The following configuration variables are currently defined:

Variable	Default Value
TET_BUILD_FAIL_TOOL	Undefined
TET_BUILD_FILE	Undefined
TET_BUILD_TOOL	Undefined
TET_CLEAN_FILE	Undefined
TET_CLEAN_TOOL	Undefined
TET_EXEC_FILE	Undefined
TET_EXEC_IN_PLACE	False
TET_EXEC_TOOL	Undefined
TET_OUTPUT_CAPTURE	False
TET_RESCODES_FILE	tet_code
TET_SAVE_FILES	Undefined

Note that a default of "Undefined" means that the variable is not provided unless it is in a test suite or a user-provided configuration variable file.

TET_BUILD_FAIL_TOOL
: This variable names a utility that will be executed if the buildtool indicates that a test case build failed.

TET_BUILD_FILE
: This variable names the file of instructions for the buildtool.

TET_BUILD_TOOL
: This variables names the buildtool.

TET_CLEAN_FILE
: This variable names the file of instructions for the cleantool.

TET_CLEAN_TOOL
: This variable names the cleantool.

TET_EXEC_FILE
: This variable names the file of instructions for the exectool.

TET_EXEC_IN_PLACE
: This variable applies only to execute mode, and is used to control where the test cases are executed. When this variable is true, the TCC executes each test case ‘in place’ in the directory containing the test case executable; when it is false, the

TCC first copies the files to a temporary directory before executing them.

TET_EXEC_TOOL
: This variable names the exectool. Normally this is left undefined, in which case the test case itself is executed. If TET_EXEC_TOOL is defined, the named program is executed with TET_EXEC_FILE and the test case name as additional arguments preceding any test case arguments.

TET_OUTPUT_CAPTURE
: Test cases not written using the TET API may write diagnostic and error messages using system output or diagnostic facilities. When this variable is true, the TCC pretends that the test case is an executable that has within it a single invocable component numbered 0 with a single test purpose. The TCC generates test case, invocable component, and test purpose start messages and then executes the test case. The output from the test case is recorded as test purpose messages (with a block number of 0). When the test case terminates, the TCC supplies a test purpose result message using the completion status of the process. It then generates invocable component and test case end messages. The default for TCC operation is false. Note that when this mode is enabled, the result file processing normally done by the TCC is disabled.

  The setting of this variable also affects build and clean modes (§4.1.1).

TET_RESCODES_FILE
: This variable indicates the name of the file which contains test suite specific result code definitions. The default for this variable is "tet_code".

TET_SAVE_FILES
: This is a comma-separated list of filenames. If files matching one of these names are found in the execution directory hierarchy after test case execution, they are transferred to the corresponding place in the saved files hierarchy.

4.1.4 Communication Variables

The TCC must be able to communicate with the other tools it executes (buildtool, cleantool, exectool, and the test cases). TCC does this via communication variables. These variables may co-exist with other resource spaces on a system. To prevent overlap with other names in the same space, implementations should reserve names beginning with "TET_". The communication variables currently defined include:

TET_CONFIG
: The pathname of the current configuration variable file.

TET_CODE
: The pathname of the current result codes definition file.

TET_ACTIVITY
: The number of activities performed thus far by the TCC. Activities include executions of buildtool, cleantool, exectool, and test cases.

TET_EXECUTE
: The pathname of the root of the execution directory hierarchy.

TET_ROOT
: The pathname of the TET root directory.

4.1.5 Result Codes

A mapping mechanism for the TCC is provided to interpret results from test suites. This mapping is contained in a results file. Upon termination of a test case, the TCC determines the action required for each result code and writes an entry in the journal. The API library ensures that the test purposes only generate TET or test suite specified results.

TET provides a default set of result codes (§7.1.3). Additional result codes may be defined on a per test suite basis. *[Requirement 2.6]*

4.1.6 Journal Entries

The TCC manages updates to the journal file. Upon completion of a test case execution, build or clean the results are transferred into the journal file. During this transfer, the TCC ensures that each executed test purpose generated one (and only one) result.

Because a test purpose may have generated other processes that may have written into the execution results file, the TCC must ensure that the information in the execution results file is ordered correctly when it is transferred into the journal. This ordering is performed using the following algorithm:

1. While the message type is not test purpose start, lines are copied from the result file to the journal.
2. Once a test purpose start message is found, it is transferred to the journal. Then all lines between that and the test purpose result message (or end of file) are ordered by:

a. Looking at the first untransferred line in the range. If the line is not a test purpose information line, it is transferred and step (a) is repeated. If it is a test purpose information line, it is transferred and the context and block identifiers are remembered.

b. The TCC then looks through the range for additional test purpose information lines with the same context and block identifiers. Those are transferred after sorting them in ascending order by sequence number.

c. When the end of the range is encountered, the TCC returns to step (a). This process is repeated until all lines in the range are transferred.

3. The test purpose result line is transferred to the journal and the TCC starts again from step 1. These steps are repeated until the execution result file is exhausted. If there is a missing test purpose result, the TCC supplies a result of `NORESULT` for that test purpose. *[Requirement 3.2]*

4.1.6.1 Future Directions

A proposed future addition is for the TCC to be capable of supporting the execution of several test cases simultaneously. In this situation, each test case places its results into a separate execution results file. The TCC captures and transfers the execution results in the order in which the test cases were specified in the test scenario. *[Requirement 5.1]*

4.1.7 Exception Conditions

4.1.7.1 Initialisation Errors

If the TCC does not locate a user specified or required input file upon test session start-up, the TCC will abort and inform the user of the error.

4.1.7.2 Failure to Build, Execute or Clean

If any specified tools (e.g. buildtool, cleantool, exectool) are not available, the TCC will generate an end message for that tool with a status of "-1" and discontinue processing of the scenario.

If, for a specified test case, the buildtool returns an abnormal termination, then the TCC will execute the optional build fail tool.

If in execute mode the TCC cannot find a test case or the test case is not executable, the TCC will indicate a test case completion status of "-1" and continue processing the scenario.

4.1.7.3 Test Session Interruption

This function is requested by using the normal system interrupt facilities. Upon receipt of the system interrupt, the TCC will order the shutdown of the currently executing test case. Test cases written to the TET API will record a TCM message in the execution results file indicating that a termination request was received, and terminate. Once the test case execution has stopped, the execution results file is captured and transferred to the journal.

4.1.7.4 Data File Corruption

Upon detection of an invalid data file format, the TCC will abort the test session and issue a diagnostic containing the pathname of the file to the journal and to the user.

4.1.7.5 Timeouts

The TCC user may specify a timeout period. Build, execution or clean-up operations which take longer than this timeout (on a per test case basis) will be interrupted by the TCC. In such a case, the TCC retrieves any execution results and transfers them to the journal. It then places a message in the journal indicating a timeout, and sets the completion status of the test case to "-2" (§7.2.1).

4.1.7.6 Lock Files

The TCC employs a locking system which prevents concurrently executing TCCs from interfering with each other’s processing of test cases. In the following description, "execution directory" means the execution directory under the alternate execution directory hierarchy if one is in use, otherwise the source directory.

In build and clean modes the TCC obtains an exclusive lock in both the source and execution directories as follows:

- A file "tet_lock" is created in the source directory, using an atomic operation which will fail if a file or directory of that name already exists. If the file cannot be created the locking operation fails.
- If an alternate execution directory hierarchy is in use, a file "tet_lock" is created in the execution directory in the same manner. If the file cannot be created, the lock is removed from the source directory and the locking operation fails.

When the build or clean has completed the lock files are removed, execution directory first.

In execute mode with TET_EXEC_IN_PLACE true, the TCC obtains an exclusive lock in the execution directory using the same method as for build mode.

In execute mode with TET_EXEC_IN_PLACE false, the TCC obtains a shared lock in the execution directory as follows:

- A directory "tet_lock" is created in the execution directory, with read and write permission for all users, using an atomic operation which will fail if a file or directory of that name already exists. If the directory cannot be created because a plain file exists, the locking operation fails. If the directory cannot be created because a directory already exists, the failure is ignored.
- A unique file is created in the directory "tet_lock". If the file cannot be created because "tet_lock" either does not exist or is a plain file, then the locking attempt is re-started.

When the execute has completed the lock is removed as follows:

- The file created when the lock was obtained is removed from the "tet_lock" directory.
- The "tet_lock" directory is removed, using an operation which will fail if the directory is not empty. Failure of this operation is ignored.

When a locking operation fails, if the user has specified a timeout period, then the TCC periodically re-tries the operation until the timeout period has elapsed. If the user has not specified a timeout period, or the timeout period elapses, then the TCC places a message in the journal indicating a locking failure, and sets the completion status of the test case, buildtool, cleantool or exectool to "-3" (§7.2.1).

4.1.8 Miscellaneous TCC Issues

4.1.8.1 Messaging

Toolkit implementations provide messages to users in their native language. This "messaging" is done in implementation defined ways, and may not be available for all languages in all implementations. Test suite authors should be aware of the languages and support for messaging provided in particular implementations, and provide equivalent functionality.

4.1.8.2 Support for Test Cases not using the defined API

It is highly desirable to support test cases written to the other test environment toolkits. Unfortunately, it is difficult to establish a mechanism which would support the many types of test cases which currently exist. Therefore, a simple mechanism is provided to map exit status codes to result codes and capture test case diagnostic output. This is done by enabling output capture mode (§4.1.3).

4.1.8.3 Support for Interactive Testing

In some environments it may be appropriate to perform tests which require interaction with users. In such environments, the TCC must not prohibit such interactive testing using normal system-provided methods. *[Requirement 3.11]* Note that in many environments interactive testing will not be possible if the TET_OUTPUT_CAPTURE configuration variable is set to true.

4.1.8.4 Combining TCC Modes

The TCC supports combining its various operational modes. If build, execute and clean modes are combined, each test case in the scenario will be individually built, executed and/or cleaned as required. If a build fails and either there is no build fail tool in use or the build fail tool terminates with abnormal completion status, then the execution and cleaning of that test case are skipped. *[Requirement 3.5]*

4.2 Application Programming Interface

The API supplies functionality to developers of test cases. This section describes that functionality in a manner independent of programming language and implementation. *[Requirement 4.4]* Through the API, developers gain support in:

- Initialising and cleaning up test cases
- Selecting invocable components and test purposes
- Insulating from the test environment *[Requirement 4.5]*
- Making journal entries
- Cancelling test purposes
- Manipulating configuration variables

- Generating and executing processes

The general flow of control for test cases built using the API is as follows:

1. The Test Case Manager (TCM) arranges for access to the configuration information. (See "Manipulating Configuration Variables" below).
2. The TCM outputs a TCM start entry to the execution results file.
3. The TCM builds a list of test purposes to be executed from the list of requested invocable components.
4. The TCM arranges for processing of asynchronous events.
5. The TCM executes the specified start-up procedure (if any).
6. The TCM commences execution of each requested invocable component in turn and outputs an invocable component start entry to the execution results file as each invocable component is started.
7. The TCM commences execution of each test purpose within an invocable component. As each test purpose is started, a test purpose start entry is output to the execution results file.

   The TCM ensures that the handling mechanism for asynchronous events is still in place before each test purpose. This is to ensure that asynchronous events do not go unnoticed because the handling mechanism has been changed by an earlier test purpose and not restored. If desired, the API may provide a facility for the test case author to disable this behaviour by setting a global flag in the start-up procedure. In this case the API programmer documentation must make clear the dangers of setting this flag.
8. If an event occurs which interrupts the processing of a test purpose, the TCM reports the receipt of the event to the execution results file as an information line. The TCM then outputs a test result of `UNRESOLVED` to the execution results file. If the event is a termination message from the TCC, then the API reports receipt of the message to the execution results file, executes the specified clean-up procedure (if any) and exits abnormally; otherwise, it continues by processing the next test purpose.
9. If a test purpose has been marked as cancelled (§4.3.5) the TCM does not attempt to execute this test purpose. Instead, it outputs an information line detailing the reason for cancellation to the execution results file. The TCM then outputs a test result of `UNINITIATED` to the execution results file and continues by processing the next test purpose.
10. Once all of the test purposes in an invocable component have been executed, the TCM outputs an Invocable Component End message and moves on to the next requested invocable component (See step 6).

11. After all test purposes for all requested invocable components have been executed, the API executes the specified clean-up procedure (if any) and exits normally.

12. If, for any reason, the test case fails to execute the requested set of invocable components or the specified start-up or clean-up procedure, the API will exit abnormally.

4.2.1 Initialising and Cleaning up Test Cases

Upon start-up, test cases written using the Toolkit API may have a test case specific start-up routine executed. Test cases may also have an optional clean-up routine executed just before test case termination.

4.2.2 Selecting Invocable Components and Test Purposes

Test Case Manager (TCM), controls sequencing of invocable components and test purposes. In order to accomplish this, it needs information about the relationship of invocable components to test purposes. Each invocable component can contain more than one test purpose, but each test purpose can be referenced only by a single invocable component. The special invocable component **all** is a reference to all test purposes. Other invocable components are requested by the TCC during test case start-up. The manner in which the relationship between invocable components and test purposes is established depends on the language binding being used. *[Requirement 4.2]*

4.2.3 Insulating from the Test Environment

The TCM provides services which automatically process asynchronous events in the system. The event mechanisms and names are implementation defined. Event selection is done via Configuration Variables. *[Requirement 4.5]*

4.2.4 Making Journal Entries

The Toolkit API journal facility support functions provide a mechanism for outputting data to the execution results file. The Toolkit API ensures that each entry in this file is written atomically, and that there is sequencing information applied to the entry (to allow for a subsequent process to re-order data that is produced from two or more concurrently executing processes started by a single test purpose).

In order to allow for the correct sequencing of information two attributes are defined: the current context and the current block. The current block is a subdivision of the current context and provides a means of ensuring contiguity, after re-sequencing, of a block of data that needed to traverse several entries. The need to traverse several entries may be caused by the limitations on the atomicity imposed by the implementation, or may be purely a matter of convenience for the test suite author.

The current context is initialised during test case start-up and can be changed only after a new process is generated. This allows the author to choose whether a number of concurrently executing test purposes should have the same context or different ones.

The current block is initialised to one at the commencement of each test purpose. The test author can increment the current block at any point during the output of entries in order to distinguish one block of data from another. Each individual entry within a block will be sequenced starting at one. Use of the journal support facilities allows data from concurrently executing test purposes to be ordered correctly by the TCC.

4.2.4.1 Entries from the API

As mentioned above, the TCM handles the sequencing of test purposes as a part of the process of executing invocable components. The sequencing mechanism will output invocable component start and end information, and test purpose start information, to the execution results file. The test author is responsible for outputting test information and test results to the execution results file.

All of the data for an entry must be transferred atomically to the execution results file. It is the responsibility of the test suite author to remain within the limitations imposed by the implementation for a single atomic write operation. TET requires that systems permit atomic writes of at least 512 bytes (§3.4).

If a test purpose executed another process that is built to the Toolkit API, and that executable is expected to generate journal messages, the test purpose must use the Toolkit API to communicate the current message context to the executed process.

4.2.4.2 Entries from Test Purposes

The API provides functionality for delivering informational messages and results from test purposes to the execution results file. These messages are in addition to those specified above, which are provided automatically by the TCM. It is expected that test cases will use this mechanism to deliver special messages to the journal or for additional reporting sequences that can be analysed by test suite specific report treatment filters. Test purposes also deliver results to the execution results file. *[Requirement 4.8]* These results are checked by the API to ensure that they have been defined by the Toolkit or by the test suite. *[Requirement 4.9]* In the event of an invalid result, the TCM delivers a message to the execution results file and sets the result to `NORESULT`. The result actions are also checked by the API. If a test purpose specifies a result for which the action is "Abort", then the TCM will immediately stop processing test purposes and exit.

Note that if a test purpose neglects to generate a result via the API, the TCM will supply a result of `NORESULT` for that test purpose.

4.2.5 Cancelling Test Purposes

In certain cases, particularly where there are a number of inter-related test purposes, the test author may wish to mark a test purpose as cancelled because of the failure of another test or because some hardware support is not available. The TCM determines the status of each test purpose prior to execution. If the TCM identifies a cancelled test purpose, then it will output a test case defined reason for the cancellation as well as a result of `UNINITIATED`. The text of the reason is provided via the API when the test suite cancels the test purpose. The test suite author can use the API to determine which test purposes are currently cancelled and the reason for cancellation.

4.2.6 Manipulating Configuration Variables

The test case controller gathers Toolkit and test suite specific configuration variable settings and provides them to test purposes via the API.

4.2.7 Generating and Executing Processes

The basic requirements for an implementation of the Toolkit state that the system allows applications to execute other applications and that these applications must have a hierarchical relationship with one another. The API provides facilities for a test purpose to generate a copy of the test case as a leaf node in this hierarchy of processes. It also allows for any node within the hierarchy of processes to replace itself with another application process. The implementation may require that these two operations are not distinct.

The TCM ensures that any generated process only performs activities associated with the current test purpose. The API also ensures that on completion, the generated process reports to the generating process the results of its execution.

The API also ensures that when a new application replaces an existing application within the hierarchy of processes, this new application can use all of the facilities within this API description in the same manner as the application which was replaced could use them. In particular the TCM ensures that the new application has access to the configuration information via the API interfaces. It further ensures that any special handling mechanisms established by the generating process are disabled in the new process. A specified routine in the generated process is then executed, and the results conveyed to the generating process via the completion status. The generating process will optionally execute a routine before analysing the completion status of the generated process. The generating process may specify a timeout for the generated process.

Explicit API support may be given to generated processes, ensuring that the above actions occur before, during, and after the execution of test purpose specified routines.

4.3 Test Suite Installer

The installation procedure for a test suite can be very complex or very simple.

It is essential that thorough installation documentation be provided by the test suite author. In particular, test suite specific configuration variables or other special information should be provided.

5. User Interface Specifications

This section defines the "command line" interfaces for tools provided in implementations of TET for various architectures. This includes the user interface to the TCC, the Report Generator, the Test Suite Installer and test cases written using the TET API. The functionality represented by these interfaces is defined in Functional Specifications (§4).

5.1 Test Case Controller

The Test Case Controller provides support for the building, execution, and clean-up of Test Scenarios. *[Requirement 1.5]* This interface is appropriate for systems which support a command shell as defined in the X/Open Portability Guide Issue 3 Volume 1 — `sh`.

5.1.1 Synopsis

`tcc -b -e -c` **[** `-a` *Alt_exec_dir* **]** **[** `-f` *Clean_cfg_file* **]**
[`-g` *Build_cfg_file* **]** **[** `-i` *Int_results_dir* **]** **[** `-j` *Journal_file* **]**
[`-s` *Scenario_file* **]** **[** `-t` *Timeout* **]** **[** `-v` *Variable=value* **]**
[`-x` *Exec_config_file* **]**
[*Test_suite* **[** *Scenario* **]]**

`tcc -b -e -c` **{** `-m` *Pattern* **|** `-r` *Pattern* **}** **[** `-a` *Alt_exec_dir* **]**
[`-f` *Clean_cfg_file* **]** **[** `-g` *Build_cfg_file* **]** **[** `-i` *Int_results_dir* **]**
[`-j` *Journal_file* **]** **[** `-s` *Scenario_file* **]** **[** `-t` *Timeout* **]**
[`-v` *Variable=value* **]** **[** `-x` *Exec_config_file* **]** *Old_journal_file* **[**
Test_suite **[** *Scenario* **]]**

5.1.2 Description

The TCC has three modes of operation, which may be executed singly or together. Each of these modes accepts options as modifiers to that mode's behaviour. The mode (with optionally modified behaviour) is then applied to the test cases and invocable components referenced by the *Scenario*. Unless otherwise specified, journal information and saved intermediate result files are placed within an implementation-defined directory under the `results` directory of the test suite selected. The TCC writes the name of the journal to standard output on startup.

5.1.3 Options

`tcc` supports the following modes.

`-b` Enable build mode.

`-e` Enable execute mode.

`-c` Enable clean mode.

These modes accept the following options.

`–m` *Pattern*

Resume from a specified point in a previous run. The argument *Pattern* specifies a comma-separated list of result code names and the letters **b**, **e** and **c**. Upon encountering an execution result corresponding to one of the specified result code names, or where **b**, **e** or **c** is specified, any build, execute or clean failure respectively, in the *Old_journal_file*, the associated invocable component and all subsequent invocable components in the scenario are built, executed and/or cleaned up, as appropriate. E.g.:

```
tcc -b -m b                 # resume building from IC which
                            # failed to build.

tcc -e -m FAIL,UNRESOLVED   # resume execution from IC which
                            # gave FAIL or UNRESOLVED result
                            # on execution.

tcc -bec -m b,e             # resume building, executing and
                            # cleaning from IC which failed
                            # to build or execute, or did not
                            # give a PASS result on execution.
```

`–r` *Pattern*

Re-run specified invocable components from a previous run. The argument *Pattern* specifies a comma-separated list of result code names and the letters **b**, **e** and **c**. Upon encountering an execution result corresponding to one of the specified result code names, or where **b**, **e** or **c** is specified, any build, execute or clean failure respectively, in the *Old_journal_file*, the associated invocable component is built, executed and/or cleaned up, as appropriate. E.g.:

```
tcc -b -r b                 # re-build all ICs which failed
                            # to build.

tcc -e -r FAIL,UNRESOLVED   # re-execute all ICs which gave
                            # FAIL or UNRESOLVED result on
                            # execution.

tcc -bec -r b,e             # re-build, execute and clean
                            # all ICs which failed to build
                            # or execute, or did not give a
                            # PASS result on execution.
```

`–a` *Alt_exec_dir*

The argument *Alt_exec_dir* specifies the pathname of the root of a directory hierarchy in which executable test cases should be placed (by the build tool in build mode), executed from (in execute mode), and cleaned from (by the clean tool in clean mode). If unspecified, the default behaviour is to execute test cases from the test suite hierarchy.

`-f` *Clean_cfg_file*

The argument *Clean_cfg_file* specifies the pathname of the file of configuration variables to use in clean mode. If unspecified this value defaults to `tetclean.cfg` in the root directory of the test suite selected.

`-g` *Build_cfg_file*

The argument *Build_cfg_file* specifies the pathname of the file of configuration variables to use in build mode. If unspecified this value defaults to `tetbuild.cfg` in the root directory of the test suite selected.

`-i` *Int_results_dir*

The argument *Int_results_dir* specifies the pathname of the root of a directory hierarchy to store intermediate result files (matching the configuration variable TET_SAVE_FILES). If unspecified the intermediate results are saved within a uniquely named directory hierarchy under the `results` directory of the test suite.

`-j` *Journal_file*

The argument *Journal_file* specifies the pathname of the file in which journal information is to be recorded. If unspecified, the journal information is stored in a file named `journal` in the root of the intermediate results hierarchy.

`-s` *Scenario_file*

The argument *Scenario_file* specifies the pathname of a scenario file. If unspecified this value defaults to `tet_scen` in the root directory of the test suite selected.

`-t` *Timeout*

The argument *Timeout* specifies the number of seconds to allow a single build, execute, or clean of a test case to continue. If unspecified, the TCC imposes no time limit.

`-v` *Variable=value*

The argument *Variable=value* is used to set the configuration variable *Variable* to *value*. Note that multiple variables may be manipulated by repeating this option with different variables names.

`-x` *Exec_config_file*

The argument *Exec_config_file* specifies the pathname of the file of configuration variables to use in execute mode. If unspecified this value defaults to `tetexec.cfg` in the alternate execution directory, or failing that, `tetexec.cfg` in the root directory of the test suite selected.

5.1.4 Operands

Old_journal_file

This operand is only used with the resume and re-run options. It indicates the pathname of the journal file that is to be searched for test failures meeting the criteria specified by the user.

Test_suite

This operand indicates the suite to which the scenario must be applied. The suite name is used by the TCC to determine the suite root directory name. If none is given on the command line, a default test suite may be deduced by the TCC according to the following rules:

1. If the current directory lies under the TET_ROOT directory hierarchy, the test suite is the component of the current directory's pathname which lies immediately below the TET_ROOT;
e.g. if the TET_ROOT is `/usr/TET` and the current directory is `/usr/TET/suite1/results/0017bec`, then the default test suite is `suite1`.

2. If the current directory lies outside of the TET_ROOT directory hierarchy, then no default may be deduced.

Scenario

This operand indicates the Test Scenario that is to be processed by the Test Case Controller. If none is given on the command line, the default scenario is `all`.

5.2 API conforming test case

Test cases written to the TET API are usable as stand-alone applications, although some set up of Communication and Configuration variables may be required. See sections 4.1.3 and 4.1.4 for more details.

5.2.1 Synopsis

`test_case` *[iclist]* ......

5.2.2 Description

The `test_case` is provided by a test suite author, and may reside anywhere in the test suite directory tree. Invocable components within the test case are selected by number.

5.2.3 Options

None.

5.2.4 Operands

iclist

A list of invocable components to be executed. This is either the special keyword "all", or a comma-separated list of single invocable component numbers or ranges of numbers separated by a dash (–). If no *iclist* is indicated, "all" is assumed. If more than one *iclist* is supplied, they are processed in the order given. If "all" occurs as a second or subsequent operand, the behaviour is unspecified.

6. Application Programming Interfaces

The section defines the interfaces that an application (test case) would use to control the execution of invocable components and test purposes and to output information and results to the execution results file (referred to as the API - Applications Programming Interface). It is expected that several different language bindings will be made to the Toolkit functionality. The following sections are normative references to language bindings for:

- The C Language on ISO 9945-1 Systems
- The Shell Language (X/Open Portability Guide Issue 3 Volume 1 — `sh`)

6.1 C Language Binding

The synopses here are described in accordance with the International C Standard ISO 9899. An ISO 9899 or Common Usage C (as defined in ISO 9945-1) conforming compiler is required to develop Test Cases using these interfaces. The interfaces in this section support all the functionality required in §4.3 of this document. In environments in which this binding is available, communication variables are placed in the Environment (as defined in ISO 9945-1).

Applications written to this language binding attach themselves to it through the following files:

`TET_ROOT/lib/posix_c/libapi.a`
: contains the support routines for test purposes.

`TET_ROOT/lib/posix_c/tcm.o`
: contains the routine *main*() and associated support routines for the sequencing and control of invocable components and test purposes.

`TET_ROOT/lib/posix_c/tcmchild.o`
: contains a *main*() routine which can be used by test suites when building processes which test purposes will launch using the *tet_exec*() interface.

`TET_ROOT/inc/posix_c/tet_api.h`
: provides a definition of `struct tet_testlist`, values for use as arguments to *tet_result*() (i.e. `TET_PASS`, `TET_FAIL`, `TET_UNRESOLVED`, `TET_NOTINUSE`, `TET_UNSUPPORTED`, `TET_UNTESTED`, `TET_UNINITIATED` and `TET_NORESULT`) and declarations/prototypes for all the 'C' API interfaces.

Each of these files should be accessed by test suites via their **buildtool** in a way which is appropriate given the available C language translation system. Test suite authors are advised to allow easy specification of alternate pathnames for these files (possible through TET configuration variables), thus improving the customisability of their suites.

Note that test cases built to this API do not require the TCC to execute. With careful setup of the communication variables (§4.1.4) test cases can be invoked from the command line (§5.4). *[Requirement 4.1]*

6.1.1 Test Case Structure and Management Functions

These functions support functionality described in Initialising and Cleaning up Test Cases and Selecting Invocable Components and Test Purposes (§4.3.1 and §4.3.2).

6.1.1.1 Synopsis

```
struct tet_testlist { void (*testfunc) (void);
                      int icref; };
struct tet_testlist tet_testlist[];
void (*tet_startup) (void);
void (*tet_cleanup) (void);
int tet_thistest;
int tet_nosigreset;
char *tet_pname;
```

6.1.1.2 Description

The `tet_testlist` array declares, in the `testfunc` element of the structure, the pointer to the function that is associated with each test purpose and, in the `icref` element of the structure, the relationship of the test purpose to an invocable component. The `tet_testlist` array is terminated by a structure with the `testfunc` element set to NULL. No other element of the array will use the value NULL for this element.

For each requested invocable component, the Test Case Manager (TCM) scans the `tet_testlist` array and executes, in order, each test purpose that is associated with that invocable component. When **all** invocable components are requested, the TCM executes all ICs for which entries are defined in the `tet_testlist` array, in ascending order. In both cases the TCM will calculate the number of test purposes that are to be executed for each requested invocable component.

The TCM will not perform any error checking on the contents of the `tet_testlist` array. It is the test author's responsibility to ensure that the array is correctly specified.

The function pointers `tet_startup` and `tet_cleanup` are set to the functions to be used for Test Case specific start-up and clean up procedures respectively. The start-up procedure is executed before the first requested invocable component and the clean-up procedure is executed on completion of the last requested invocable component. Note that if either of these pointers is set to `NULL`, the TCM will not attempt to call the respective function.

The TCM will be provided as the *main*() routine to the test case program and will contain an external declaration to the `tet_startup` and `tet_cleanup` function pointers and to the `tet_testlist` array.

The `tet_thistest` variable contains the sequence number (starting at 1) of the element in the `tet_testlist` array that is associated with the currently executing test purpose. During execution of the start-up and clean-up functions, `tet_thistest` is set to zero.

The `tet_nosigreset` variable controls whether the TCM reinstates signal handlers for unexpected signals before each test purpose. The default value of zero means that signal handlers will be reinstated before each test purpose, to ensure that unexpected signals do not go unnoticed if an earlier test purpose installed a local handler but did not restore the original handler. If tet_nosigreset is set to a non-zero value in the start-up function called via tet_startup, then signal handlers will be left in place between test purposes. In test cases where stray signals constitute a test failure, it is recommended that tet_nosigreset is left with its default value of zero. This is because even if test purposes contain code to restore the signal handling, this code will not be executed if an unexpected signal arrives and the TCM skips to the start of the next test purpose.

The `tet_pname` variable contains the process name as given on the test case command line.

6.1.2 Insulating from the Test Environment

The following Configuration Variables are used by the C Language TCM to help determine which events should be handled for the Test Case, and which should be passed through. They are used by the TCM to support functionality specified in Insulating from the Test Environment (§4.3.3).

TET_SIG_IGN
: This variable defines (by comma-separated number) the set of signals that are to be ignored during test purpose execution. Any signal that is not set to be ignored or to be left (see TET_SIG_LEAVE below) with its current disposition, will be caught when raised and the result of the test purpose will be set to `UNRESOLVED` because of the receipt of an unexpected signal. A test purpose may undertake its own signal handling as required for the execution of that test purpose; the disposition of signals will be reset after the test purpose has completed, unless the global variable `tet_nosigreset` is non-zero. The TCM needs to know how many signals the implementation supports in order to set up catching functions for these signals.

TET_SIG_LEAVE
: This variable defines (by number) the set of signals that are to be left unchanged during test execution. In most cases this will mean that the signal takes its default action. However, the user can change the disposition of the signal (to ignore) before executing the TCC if this signal is to remain ignored during the execution of the test purposes.

The implementation should not allow "standard" signals to be set to be ignored or left unchanged, as this may pervert test results.

6.1.3 Making Journal Entries

These functions support functionality described in Making Journal Entries (§4.3.4).

6.1.3.1 Synopsis

```
void tet_setcontext(void);
void tet_setblock(void);
void tet_infoline(char *data);
void tet_result(int result);
```

6.1.3.2 Description

The *tet_setcontext*() function sets the current context to the value of the current process ID. A call to *tet_setcontext*() should be made by any application which executes a *fork*() to create a new process and which wishes to write entries from both processes. The call to *tet_setcontext*() must be made from the child process, not from the parent.

The *tet_setblock*() function increments the current block ID. The value of the current block ID is reset to one at the start of every test purpose or after a call to *tet_setcontext*() which altered the current context. The sequence ID of the next entry, a number which is automatically incremented as each entry is output to the execution results file, is set to one at the start of each new block.

The *tet_infoline*() function outputs an information line to the execution results file. The sequence number is incremented by one after the line is output. If the current context and the current block ID have not been set, the call to *tet_infoline*() causes the current context to be set to the value of the calling process ID and the current block ID to be set to one.

The *tet_result*() function sets the result to `result`. This result is output to the execution results file by the TCM upon test purpose completion. This ensures that all informational messages are written out before the test purpose result, and that there is one (and only one) result generated per test purpose.

6.1.4 Cancelling Test Purposes

These functions support functionality described in Cancelling Test Purposes (§4.3.5).

6.1.4.1 Synopsis

```
void tet_delete(int test_no, char *reason);
char *tet_reason(int test_no);
```

6.1.4.2 Description

The function *tet_delete*() marks the test purpose specified by `test_no` as cancelled and will output `reason` as the reason for cancellation on the information line which is output whenever the test driver attempts to execute this test purpose. The argument `test_no` is the sequence number (starting at 1) of the corresponding element in the `tet_testlist[]` array. If the requested `test_no` does not exist no action is taken. If the requested `test_no` is already marked as cancelled the reason is changed to `reason` and the test purpose remains marked as cancelled. If the `reason` is set to `(char *) NULL` then the requested `test_no` is marked as active, this allows previously cancelled test purposes to be re-activated. If `reason` is not NULL it must point to static data, as the calling function will have terminated when the reason string is accessed by the TCM. This function cannot be called from a child process.

The function *tet_reason*() returns a pointer to a string which contains the reason why the test purpose specified by `test_no` has been cancelled. If this test purpose is not marked as cancelled, or does not exist, a value of `(char *) NULL` is returned.

6.1.5 Manipulating Configuration Variables

The functions in this section support functionality specified in Manipulating Configuration Variables (§4.3.6).

6.1.5.1 Synopsis

```
char *tet_getvar(char *name);
```

6.1.5.2 Description

The function *tet_getvar*() retrieves the setting of the configuration variable `name` and returns a pointer to that setting. Implementations of this function are assumed to be re-entrant.

Note that if a variable is defined but has no value assigned, *tet_getvar*() returns a pointer to an empty string. If a requested variable is undefined, *tet_getvar*() returns a NULL pointer.

6.1.6 Generating and Executing Processes

These functions support the functionality described in Generating and Executing Processes (§4.3.7).

6.1.6.1 Synopsis

```
int tet_fork(  void (*childproc)(void),
               void (*parentproc)(void), int waittime,
               int validresults);
int tet_exec(  char *file, char *argv[], char *envp[]);
extern pid_t tet_child;
```

6.1.6.2 Description

The *tet_fork*() function will create a new process which is a copy of the calling process and will modify the signal disposition in the newly created process such that any signals that were being caught in the parent process are now set to their default values in the child process. The child process will then commence execution of the routine *childproc*() and upon completion of this routine will terminate with an exit code that indicates to the parent the correctness of execution of this routine. If the routine *parentproc*() is not set to NULL, this routine will be executed by the parent process before the exit code of the child process is waited for. On completion of the *parentproc*() routine, if any, the exit code returned by the child process will be examined by masking off the bits which are set in `validresults`. If the result is zero, *tet_fork*() assumes that this was a legal (or expected) termination code. If not, it assumes that the child process completed with an unexpected result and an error has occurred. This unexpected result is reported to the execution results file. The *tet_fork*() interface will return -1 if the result of the child process was invalid or the valid result code if the result of the child process was one of the valid results. When *tet_fork*() returns -1 it reports the nature of the error using *tet_infoline*() and sets the test result code to `UNRESOLVED` by calling *tet_result*(). If `waittime` is not set to zero, the parent process will ensure that the child process does not continue to execute for more than `waittime` seconds after the completion of the routine `parentproc`.

The *tet_exec*() function may be called from a *childproc*() routine of a child process generated by a call to *tet_fork*(). The *tet_exec*() function will pass the argument data as specified by `argv` and the environment data specified by `envp` to the process specified by `file`. The usage of the *tet_exec*() is equivalent to that of the ISO 9945-1 *execve*() function, except that the API can add arguments and environment data that are to be interpreted by the driver of the executed `file`. If *tet_exec*() is called without first calling *tet_fork*(), the results are undefined. The *tet_fork*() function makes calls to *tet_setcontext*() in the child and *tet_setblock*() in the parent to distinguish output from the child and from the parent before, during, and after execution of the `parentproc` routine.

The global variable `tet_child` is provided for use in the `parentproc` routine called from *tet_fork*(). It is set to the process ID of the child.

6.1.7 Executed Process Functions

These functions are to be used by processes executed through the *tet_exec*() function. They support functionality described in Generating and Executing Processes (§4.3.7).

6.1.7.1 Synopsis

```
int tet_main (int argc, char *argv[]);
int tet_thistest;
char *tet_pname;
```

6.1.7.2 Description

The function *tet_main*(), supplied by the test suite developer, is called by the *main*() function of the TET-supplied child process controller `tcmchild.o`. Prior to calling *tet_main*(), `tcmchild.o` will set the `tet_thistest` variable to the number associated with the test purpose in the original test case. This value should not be changed by the executed process. Upon completion of the *tet_main*() routine the return value from this function will be returned to the process which called the *tet_fork*() function. The value returned from *tet_main*() should, in most cases, match one of the valid result values specified on the call to the *tet_fork*() function. The current context is preserved from the calling process and the current block is incremented by one before *tet_main*() is called.

The `tet_pname` variable contains the process name as given in argv[0].

6.2 Shell Language Bindings

The interfaces in this section support all the functionality required in §4.3 of this document. Implementations supporting this language binding must support the command shell `sh` as defined in the X/Open Portability Guide Issue 3 Volume 1. In such implementations, the communication variables are placed in the Environment.

Support for this language binding is provided through two shell language source files. The file `TET_ROOT/lib/xpg3sh/tcm.sh` contains the support routines for the sequencing and control of invocable components and test purposes (the shell TCM). The file `TET_ROOT/lib/xpg3sh/tetapi.sh` contains the support routines for test purposes (the shell API). These files must be "sourced" into an executing shell language test purpose using the **dot** `( . )` shell built-in command. Sourcing the shell TCM also automatically sources the shell API.

Note that test cases built to this API do not require the TCC to execute. With careful setup of the communication variables §4.1.4) the test case can be invoked from the command line (§5.4). *[Requirement 4.1]*

6.2.1 Test Case Structure and Management Functions

These functions support functionality described in Initialising and Cleaning up Test Cases and Selecting Invocable Components and Test Purposes (§4.3.1 and §4.3.2).

6.2.1.1 Synopsis

```
iclist={blank-separated list of invocable components}
{icname}={blank-separated list of test purposes}
tet_startup={start-up function}
tet_cleanup={clean-up function}
tet_thistest
```

6.2.1.2 Description

The start-up routine, clean-up routine and each of the test purposes are to be implemented by the test author as either shell functions or as separate executable shell scripts. These shell functions or scripts will be called by the shell TCM according to the requested set of invocable components. The `iclist` is provided by the test suite author, and contains a blank-separated list of invocable component names. These invocable component names are formed by prefixing each invocable component number with the letters **ic**.

When an invocable component is requested by the TCC, the shell TCM executes each name in the associated list of test purposes. Each of the test purposes will be executed in a sub-shell with the appropriate signal handling being applied to the sub-shell.

The TCM will not perform any explicit error checking on the contents of a list of test purposes. It is the responsibility of the test author to ensure that the names reference shell functions or executable shell scripts.

The shell variables `tet_startup` and `tet_cleanup` are set to refer to the shell function or script to be used for Test Case specific start up and clean-up procedures, respectively. The start-up procedure is executed before the first requested invocable component and the clean-up procedure is executed on completion of the last requested invocable component. If no start-up or clean-up is required, the `tet_startup` and `tet_cleanup` variables may be left unset or set to an empty string.

The TCM and API are provided as shell scripts which must be sourced by the test suite author immediately after the shell variables for `tet_startup`, `tet_cleanup`, `iclist`, each of the `icnames`, and any shell functions have been defined. (The shell script is sourced by use of the **dot** `( . )` shell built-in command). Note that if a test purpose is written as a separate shell script, that script must source the shell API in order to have access to those support routines.

The `tet_thistest` shell variable contains the name of the test purpose, as specified in the `icname` variable.

6.2.2 Insulating from the Test Environment

The following Configuration Variables are used by the Shell TCM to help determine which events should be handled for the Test Case, and which should be passed through. They are used by the TCM to support functionality specified in Insulating from the Test Environment §4.3.3).

TET_SIG_IGN
: This variable defines (by comma-separated number) the set of signals that are to be ignored during test purpose execution. Any signal that is not set to be ignored or to be left (see TET_SIG_LEAVE below) with its current disposition, will be caught when raised and the result of the test purpose will be set to `UNRESOLVED` because of the receipt of an unexpected signal. A test purpose may undertake its own signal handling as required for the execution of that test purpose, the disposition of signals will be reset after the test purpose has completed. The API needs to know how many signals the implementation supports in order to set up trap statements for these signals.

TET_SIG_LEAVE
: This variable defines (by number) the set of signals that are to be left unchanged during test execution. In most cases this will mean that the signal takes its default action. However, the user can change the disposition of the signal (to ignore) before executing the TCC if this signal is to remain ignored during the execution of the test purposes.

The implementation should not allow "standard" signals to be set to be ignored or left unchanged, as this may pervert test results.

6.2.3 Making Journal Entries

The functions in this section implement functionality detailed in Making Journal Entries (§4.3.4).

6.2.3.1 Synopsis

```
tet_setcontext
tet_setblock
tet_infoline data ...
tet_result result
```

6.2.3.2 Description

The `tet_setcontext` shell function sets the current context to the value of the process ID of the calling shell. The current context is stored in the environment variable TET_CONTEXT which is marked for export. This allows the context to be passed to subsequent processes by using this environment variable. The `tet_setcontext` command should be executed by any application which executes a background subshell and which wishes to write entries from both processes. The `tet_setcontext` command must be executed from the child process, not from the parent. The parent should call `tet_setblock` as appropriate to distinguish its output before, during and after execution of the child.

The `tet_setblock` shell function increments the current block ID. The value of the current block ID is reset to one at the start of every test purpose and after a call to `tet_setcontext` which altered the current context. The sequence ID of the next entry is set to one at the start of each new block. The current block ID is stored in the variable TET_BLOCK which is marked for export.

The `tet_infoline` shell function outputs an information line to the execution results file. The sequence number is incremented by one after the line is output. If the current context and the current block ID have not been set, the call to `tet_infoline` causes the current context to be set using the value of the calling process ID and the current block ID to be set to one. Note that `tet_infoline` does not process backslash escapes like the shell `echo` built-in command. If more than one argument is passed to `tet_infoline`, then they will be output separated by a space character.

The `tet_result` shell function outputs a test result to the execution results file. This result is output to the execution results file by the TCM upon test purpose completion. This ensures that all informational messages are written out before the test purpose result, and that there is one (and only one) result generated per test purpose.

6.2.4 Cancelling Test Purposes

These functions support functionality described in Cancelling Test Purposes (§4.3.5).

6.2.4.1 Synopsis

```
tet_delete test_name reason
tet_reason test_name
```

6.2.4.2 Description

The shell function `tet_delete` marks the test purpose specified by `test_name` as cancelled and will output `reason` as the reason for cancellation on the information line which is output whenever the test driver attempts to execute this test purpose. The argument `test_name` matches the name which is used to call this test purpose. If the requested `test_name` does not match the name of a test purpose no action is taken. If the requested `test_name` is already marked as cancelled the reason is changed to `reason` and the test purpose remains marked as cancelled. If `reason` is an empty string then the requested `test_name` is marked as active, this allows previously cancelled test purposes to be re-activated.

The shell function `tet_reason` outputs, to the standard output device, the reason why the test purpose specified by `test_name` has been cancelled and exits with an exit code of 0. If this test purpose is not marked as cancelled, or does not match the name of a test purpose, there is no output, to the standard output device, and the function exits with an exit code of 1.

6.2.5 Manipulating Configuration Variables

There is no explicit Shell interface to support this functionality. The API will ensure that the configuration information is available to the test purposes as shell variables marked **readonly**. Each of these shell variables can be accessed using the normal shell mechanisms.

6.2.6 Generation and Execution of Processes

There is no explicit Shell interface. The API ensures that the configuration information and the `tet_thistest` shell variable are available as shell variables. The ability to use parentheses to generate a subshell environment allows these variables to be inherited when a subshell is generated. The only facilities that are not provided in the shell are the ability to timeout a subshell process and the examination of the exit code from the subshell. The shell provides facilities to accomplish these tasks in a relatively straightforward manner and this is considered to be an issue for the application programmer rather than for the API.

6.2.7 Executed Process Support

Shell scripts which are executed by a test case written to this API should source (using the **dot** (.) shell built-in command) the shell API to include the necessary support routines. Note that this will not provide TCM functions (like signal handling and test purpose sequencing). Executed processes which need this type of support should be test cases in their own right.

7. Data File Formats

This section defines the format of Toolkit portable data files. These data files (provided by the Toolkit, test suite authors, and Toolkit users) are portable to all implementations of the Toolkit. Note that these formats may be extended by conforming implementations.

7.1 Input Files

All Toolkit input files use a similar set of construction rules:

- Data elements start in column 1 and are terminated with an *end-of-line* character.
- Comments start with a # in column 1, and later terminated with an *end-of-line* character.
- Blank lines (lines consisting only of *blanks* and the *end-of-line* character) are ignored.

7.1.1 Configuration Variables Files

There is a single configuration file per executing test scenario per execution mode. This configuration file is found, by default, in the root directory of the appropriate test suite. By default this file is called `tetbuild.cfg`, `tetexec.cfg`, or `tetclean.cfg` for each of build, execute, and clean modes respectively. The names and locations of these files can be changed through options to the TCC. See the Test Case Controller Interface Specification (§5.1) for further details.

In a configuration variable file, variables are defined with lines of the form *name*=*value*. The *name* may consist of any sequence of alphanumeric characters and the underscore character which does not start with a numeric character. The *value* may consist of any sequence of characters.

Note that names starting with the sequence "TET_" are reserved for implementations of this specification.

Example:

```
# Example configuration variables settings.
TET_BUILD_TOOL=make -f
TET_BUILD_FILE=Makefile
```

7.1.2 Scenario Files

Scenario files assign user visible names to one of more invocable components. The default scenario file is called `tet_scen` and is located in the root directory of each test suite. This file must at least define a scenario named **all**, which should cause execution of every invocable component in the suite. Test cases and "include files" are referenced in scenarios via pathnames. In build and clean modes, and in execute mode when there is no alternate execution directory, test case pathnames are relative to the test suite root directory, and may only reference files within the test suite hierarchy. In execute mode when there is an alternate execution directory, test case pathnames are relative to that directory and may only reference files within the alternate execution directory hierarchy. Include file pathnames are always relative to the test suite root directory.

The scenario file is made up of multiple data elements. Each data element consists of a name and one or more instructions. Scenario file data elements may continue beyond the end of a line. Continuation of a data element is indicated by beginning subsequent lines beyond column 1. Note that comments (indicated by a # in column 1) do not indicate the end of a data element.

The name is built of alphanumeric characters, and may be up to 31 characters in length.

A scenario file data element may contain any of the following instructions:

1. Invocable component execution. These specify the test cases and invocable components to be executed. Multiple invocable components within a single test case may be specified in a single instruction. Invocable component names consist of a leading slash, the pathname of a test case (see above), and optional comma-separated invocable component numbers within braces ({,}) (e.g. `/tests/readtests/test.exe{1,2,3}`).

2. Journal message output. This instruction causes a message to be placed into the journal with the type `scenario information`. These lines could be used by a specific report treatment filter to establish check points in a report. Journal messages must be surrounded by quotation marks (") and may not contain quotation marks (e.g. `"This is a Journal Message"`).

3. Invocable component inclusion. Instructions of this type reference a set of invocable components in another file. The included invocable components are executed synchronously. This facility can be used to reduce the complexity of scenario files. Include directives consist of the colon-delimited term **include** and the pathname to include (e.g. `:include:/tests/readtests/testlist`).

Files referenced by include directives consist of entries, each of which is either a journal message (see above) or the pathname of a test case with optional invocable components (e.g. `/tests/readtests/test1{1,3}`).

Example:

```
# Example scenario file
readtest /tests/readtests/test1{1,3}
```

```
writetest /tests/writetests/test1{1,2,3}
all /tests/readtests/test1
    "Read tests completed"
    /tests/writetests/test1
    "Write tests completed"
```

7.1.2.1 Future Directions

The following scenario file features are proposed future additions:

1. Invocable component iteration. These indicate that a set of invocable components (defined in the referenced file) are to be executed synchronously, but repeatedly. Iteration directives consist of colon-delimited terms **repeat,#** and the pathname to include, where # is the number of times to repeat the included ICs (e.g. `:repeat,10:/tests/readtests/testlist`).
2. Invocable component simultaneous execution. These indicate that a set of invocable components (defined in the referenced file) are to be executed asynchronously. They consist of a colon-delimited term **group[,#]** and the pathname of ICs to group, where # is an optional repeat count (see iteration, above) (e.g. `:group,10:/tests/readtests/testlist`).

7.1.3 Results Code Definitions

The Toolkit and test suites both supply result code definitions. Test suite supplied result code definitions are stored in files called `tet_code` in the test suite root directories. *[Requirement 5.2]* These files are made up of lines which have single elements. These elements contain up to three blank-separated fields, defined as follows:

1. The result code. This is a non-negative decimal integer between 0 and 127, inclusive. Result codes from 0 to 31 (inclusive) are reserved for use by the TET implementation. The remainder are available for specification by the test suite author.
2. The name of this result. This is a double quote-delimited field which can contain any sequence of alphanumeric characters and blanks.
3. The action to take when this result is encountered. This is an indication of what the TCC and TCM should do when the result is returned by a test purpose. Legal values are `Continue` and `Abort`. The default value for this field is `Continue`.

The following is an example result codes definition file. It contains those result codes provided by TET.

```
0   "PASS"          Continue
1   "FAIL"          Continue
2   "UNRESOLVED"    Continue
3   "NOTINUSE"      Continue
4   "UNSUPPORTED"   Continue
5   "UNTESTED"      Continue
6   "UNINITIATED"   Continue
7   "NORESULT"      Continue
```

7.2 Output Files

7.2.1 Journal and Result Files

Result files are written by test cases, buildtools and cleantools when run with output capture mode disabled. These result files are then transferred into the journal file by the TCC. The format of lines in these files is identical. The total length of a journal line must not exceed 512 bytes. Each journal line is made up of a message type, the parameters for that message, and a message area (the format of which is unconstrained). Each message may have zero or more parameters associated with it. These parameters (strings or integers represented by no more than ten decimal digits) are blank-separated and are contained between vertical bars. Possible parameters include:

1. The TCC activity number (Activity). This number is incremented once for each activity performed by the TCC (build, execute and clean-up are each considered individual activities).
2. The invocable component number (ICnumber).
3. The test purpose number (TPnumber). This number uniquely identifies the test purpose within a test case.
4. The test purpose count (TPcount). This is the number of test purposes that make up an invocable component.
5. The message context (Context). This field represents the process that generated the journal line. Its make up is implementation defined. Implementations are required to ensure that there are no repeated process identifiers during the life of a test purpose.
6. The message block number (Block). This number is set to one at the start of each test purpose or new context, and is incremented once when the test purpose requests it. This number, along with the process identifier and message number (below) is used by the TCC to order the data in an execution result file prior to transferring that file into the journal.
7. The message sequence number (Sequence). This number is set to one at the start of each block, and is incremented once for each message written to the result file.
8. The current time (Time). Times are given using the notation HH:MM:SS with a 24 hour clock.
9. The current date (Date). Dates are given using the notation YYYYMMDD, e.g. 19910610 for 10th June 1991.
10. A test case name (TestCase). This is the test case name as given in the scenario file.
11. A pathname (Pathname). The full pathname of a file.
12. A mode (Mode). Legal modes are Build (0), Execute (1), and Clean-up (2).

13. A completion status (Status). This is the value returned to the TCC by the operating system after execution of another process. Values less than zero are reserved by TET.
14. An invocable component count (ICcount). This is the number of invocable components executed in each test case (expected or actual as specified).

In the list below each message type is listed, followed by a line indicating the number associated with that message and the required parameters, followed by a textual description of the message and its uses.

Test Case Controller Start

```
0|Version Time Date|who
```

This message is generated by the TCC at the beginning of each TCC execution. The parameters include the version of the TCC used, the time at which TCC execution started, and the current date. The message area contains information about who ran the test.

Test Case Start

```
10|Activity TestCase Time|IC list
```

A test case start message is generated by the TCC for each test case executed in a scenario. The parameters are the sequence number of this TCC activity, the pathname of the test case, and the time the test was started. If the corresponding scenario line contains a list of invocable components to be executed, then that IC list is included in the text area of the message.

Test Case Manager Start

```
15|Activity Version ICcount|
```

The TCM writes this message during its initialisation. The parameters include the sequence number of this TCC activity, the version of the TCM being used, and the number of invocable components to be executed. The number of invocable components should be used by report treatment filters which audit test results to ensure that all ICs were correctly executed.

Configuration Start

```
20|Pathname Mode|
```

Configuration start messages are placed in the journal by the TCC at test session startup. Parameters are the pathname of the configuration file being referenced, and the mode to which this configuration information applies.

Configuration Variable Setting

```
30||variable=value
```

A configuration variable setting line is written to the journal by the TCC for each variable set for the current mode. The message area is of the form "variable=value".

Configuration End

`40||`

A configuration end message is placed in the journal by the TCC to indicate the end of configuration options for a particular mode.

Test Case Controller Message

`50||`

The TCC generates messages when it encounters execution problems. The message area gives an indication of the nature of the problem.

Scenario Information

`70||text`

The TCC generates a scenario information line when it encounters a journal message in the scenario file being processed. The text of the message is included in the message area.

Test Case End

`80|Activity Status Time|`

A test case end message is generated by the TCC after each test case completes execution. The parameters are the TCC activity number, the time the execution completed and an indication of the TC’s completion status.

Operator Abort

`90|Time|`

The TCC generates an operator abort messages when its execution is interrupted by the user.

Captured

`100|Activity|text`

The TCC generates a captured message for each line of output captured when the output capture mode is enabled. The parameter is the TCC’s activity counter.

Build Start

`110|Activity TestCase Time|`

A build start message is generated by the TCC prior to buildtool execution. The parameters are the TCC activity counter, the pathname of the test case being built and the time the build started.

Build End

```
130|Activity Status Time|
```

A build end message is generated by the TCC upon completion of buildtool execution. It contains the TCC activity number, an indication of the completion status of buildtool and the time of that completion.

Test Purpose Start

```
200|Activity TPnumber Time|
```

The TCM generates a test purpose start message for each test purpose executed. The message contains the TCC activity number, the test purpose number, and the time execution started.

Test Purpose Result

```
220|Activity TPnumber Result Time|
```

The TCM generates a result message for each test purpose executed. This result is usually supplied by the test purpose, although it is supplied by the TCM in the event a test purpose does not specify a result. Legal results are specified in Result Code Definitions (§7.1.3). Parameters for this message are the TCC activity number, the test purpose number, the result code, and the time execution completed.

Clean Start

```
300|Activity TestCase Time|
```

The TCC writes a clean start message to the journal before invoking cleantool for a given test case.

Clean End

```
320|Activity Status Time|
```

After a clean operation on a test case is complete, the TCC records the TCC activity number, the completion status, and the time at which the operation was completed.

Invocable Component Start

```
400|Activity ICnumber TPcount Time|
```

The TCM generates an invocable component start message for each invocable component that is executed during a test case run. This message contains the TCC activity counter, the invocable component number, number of test purposes to be executed, and the time invocable component execution started.

Invocable Component End

```
410|Activity ICnumber TPcount Time|
```

The TCM generates an invocable component end message for each invocable component that is completed. The message contains the TCC activity counter, the invocable component number, the number of test purposes actually executed and the time of the invocable component completion.

Test Case Manager Message

```
510|Activity|text
```

If the TCM/API encounters a problem, it will be written to the journal as a TCM message. The parameters indicate the TCC activity counter. The message area contains a brief description of the problem.

Test Case Information

```
520|Activity TPnumber Context Block Sequence|text
```

When a test case outputs information to the execution results file, it is recorded as test case information. The message specified by the test case is in the text area of this line.

Test Case Controller End

```
900|Time|
```

When the TCC completes sequencing of the scenario, it generates a TCC end message. This message indicates the time execution completed.

Appendix A - Contacts

The following constitute the Management Committee of the Project:

Open Software Foundation

John Morris
Director, Software Engineering
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142 USA

+1 617-621-8739
+1 617-225-2782 (fax)
jsm@osf.org

UNIX International

Shane McCarron
Project Manager, Technology and Product Planning
UNIX International
20 Waterview Blvd.
Parsippany, NJ 07054 USA

+1 201-263-8400x232
+1 201-263-8401 (fax)
mccarron@uiunix.ui.org

X/Open

James de Raeve
Conformance Strategy Manager
X/Open Company, Ltd.
Apex Plaza
Forbury Road
Reading RG1 1AX
Berkshire ENGLAND

+44 (0) 734 508311
+44 (0) 734 500110 (fax)
j.deraeve@xopen.co.uk

Appendix B - Requirements

Participants

The Project asked for requirements input from a number of organisations. These organisations represented Government, Development, and Conformance Testing interests. Through their contributions, the Project believes that it has received the broad input necessary to make the work valid throughout the testing community.

Approach

In the following pages, the requirements submitted have been categorised and, where possible, combined into single instances (many requirements were repeated by a number of the participants). The requirements have been analysed and prioritised by the Project. In making this summary, it is hoped that the essence of the original requirements has been maintained

The requirements are further categorised by their relative importance. The categories are:

Parenthood

> The requirement is a base assumption of any testing support software, and is therefore automatic.

Essential

> The requirement must be in any Toolkit implementation in order to satisfy the goals of the Project.

Nice

> The requirement is useful, but is not a base requirement. If provided, this is a "value-added" extra. Requirements categorised as "Nice" which are not in the initial implementation are expected to be provided in a future version.

Future

> The requirement is one which the Project agrees with, and may include in future implementations.

Never

> The requirement is not useful to a broad enough audience to be included in a Toolkit implementation. However, the requirement does represent functionality that some test writers may find useful in specific situations.

A notation such as "Essential+" means that the requirement is "essential", but that it has aspects that are "nice".

B.1 Overall Requirements

B.1.1 Portability

Essential

The Toolkit specification must be portable across a wide variety of systems. In particular, it must be portable to systems that are based upon UN*X. The minimum requirements of the Toolkit must be specified well enough to implement on a new system.

The Toolkit must conform to widely accepted industry standards whenever those are available.

B.1.2 Availability of Implementation and Specifications

Essential

The implementation and specification of this work must be freely available, and must explicitly permit the distribution of derivative works without royalty or restrictions. Derivative works that are inconsistent with the Toolkit specifications must indicate that they have been modified, and clearly state that they are not compatible with the Toolkit specification.

B.1.3 Intuitive Structure

Parenthood

The specifications must provide for intuitive, well documented, and consistent interfaces. This means, for instance, that the look and feel of test execution via the user interface should appear to be the same, regardless of the structure of the underlying Test Cases.

B.1.4 Documentation

Essential

User and Programmer Manuals must be provided. In addition, there must be a Style Guide that outlines recommended test development practices.

B.1.5 User Interfaces

Essential

A user interface to the Toolkit must be provided. In environments that support "command line" style interfaces, such an interface to the Toolkit must be provided.

B.1.6 Development Environment

Parenthood

In addition to the testing environment, sufficient structure must be provided such that Test Cases can be developed under the Toolkit. This includes the requirement that the Toolkit does not preclude the use of a source code control mechanism (e.g. RCS and/or SCCS).

B.2 Generic Tool Requirements

B.2.1 Installation of Test Suites

Essential

The installation tool must allow the Test Suite to have maximum control of its environment and configuration during installation. This means that the Installation Tool must impose the minimum necessary structure on Test Suites.

B.2.2 Configuration of Test Suites

Essential

User configuration of the Test Case Controller and Test Suites should allow for the specification of system specific details (e.g. additional libraries required for compilation, different compiler names, alternate header hierarchies)

Run time configuration should be changeable. Further, configuration should be able to be set per user, per test-suite, or globally.

Configuration information should be accessible to the user.

B.2.3 Internationalisation

Essential

A Toolkit implementation must allow international environments. This includes the selection of user locale based on industry standard mechanisms.

B.2.4 Binary Testing

Essential

The Toolkit must be capable of executing Test Cases that are distributable in executable form in appropriate environments.

B.2.5 Documentation Extraction

Nice +

The Tools must be able to extract the purpose and strategy for a given Invocable Component. This information might be used for user education, or by report generators.

B.2.6 Result Codes

Essential

The Toolkit must be able to extend its base set of result codes with a Test Suite specific set.

B.2.7 Report Generation

Essential

The Toolkit must provide report generation facilities.

B.3 Test Case Controller Requirements

B.3.1 Selective Execution

Essential

The Test Case Controller must allow a user to select which Invocable Components are to be run.

The names of Invocable Components must be made available to the Toolkit by Test Suites. It is likely that Test Suites will come pre-configured with popular sequences of Invocable Components.

B.3.2 Test Interruption

Essential

It must be possible to interrupt execution of a Test Case, and optionally to halt execution of the Test Scenario. An "added value" extra would be to support interrupts based on time interval or iteration count.

B.3.3 Restart of Interrupted Tests

Nice

An interrupted Test Scenario must be able to be re-started at the same or next Invocable Component in the Test Scenario. Such control must be available at the time of interruption, or at some time in the future. Test Scenarios must be interruptible by the operator.

B.3.4 Network Testing

Essential

The Test Case Controller must not preclude the execution of Invocable Components on connected, capable systems.

Future

The Test Case Controller should provide facilities that can take advantage of parallelism and multiple, simultaneous executions on connected, capable systems. Such facilities must be selectable by the user and/or Test Suite.

B.3.5 Test Case Build/Run

Essential

The Test Case Controller must allow the user to request that Test Cases be built before execution. A user option must be provided so that that Test Cases are built, run, and cleaned up before another Test Case is started.

The Test Case Controller must be configurable in such a way that Test Case source files are separated from the actual tests being run. It must be possible to keep a clean copy of the Test Cases at all times.

B.3.6 Selection Criteria

Essential

The criteria for selecting which tests are to be executed by the Test Case Controller must be readily definable by Toolkit users or by the system.

B.3.7 Isolation

Essential

The Test Case Controller must not prevent individual Test Cases from being executed in parallel. Each Test Case execution should be protected from like executions.

The Test Case Controller must provide cleanup of Test Case space, so that Test Cases do not corrupt each other, or induce inter-test dependencies.

B.3.8 Black-Box Testing

Parenthood

The Test Cases must communicate with the Test Case Controller only through the API as defined in the Toolkit specifications. The Test Case Controller cannot have any knowledge of the internals of an individual Test Case.

B.3.9 Audit Trails

Essential

In addition to providing Test Cases with Journal facilities, the Test Case Controller must mark in the Journal the start and finish of each Test Case execution.

B.3.10 Sequencing

Essential

The Test Case Controller must ensure that Invocable Components are processed in the sequence defined in the Test Scenario.

B.3.11 Interactive Testing

Essential

The Test Case Controller must not use the system in such a way that executing Test Cases are prevented from interacting with the operator.

B.4 API Requirements

B.4.1 Stand-Alone Tests

Essential

The API must allow a test to be built in such a way that it can be executed without the Test Case Controller. Compilation of stand-alone tests should be possible with a minimum of difficulty.

B.4.2 Test Structure Support

Parenthood

The API must allow test developers to write tests in a well structured manner.

B.4.3 Dynamic Configuration

Essential

The API must allow dynamic configuration of run time parameters. In this way, stand-alone executables can test in different ways without re-compilation.

B.4.4 Language Independence

Essential

The specification of the API must be such that it can be extended to other languages.

B.4.5 Exception Handling

Essential

The API must provide exception handling mechanisms to deal with unusual system events that occur while Test Cases are executing.

B.4.6 Test Synchronisation

Future

The API should provide facilities that will assist in synchronising multiple processes associated with a single Test Case.

B.4.7 Test Progress Tracking

Nice

The API must provide a mechanism that allows users to view the progress of Invocable Component execution.

B.4.8 Result Code Journal Information

Essential

The API must provide functionality that will record messages and result codes in a journal. These messages must be delivered to the journal synchronously.

B.4.9 Result Code Enforcement

Essential

The API must provide a mechanism that allows Test Case authors to insert support for Result Code checking and enforcement.

B.4.10 Resource Requirements

Essential

The API must document statically-used resources. The API must also provide a run time method to determine the dynamically allocated resources.

B.5 Journal Requirements

B.5.1 Comprehensive Reporting

Essential

The journal-recording mechanism must provide information which includes, at the very least, Test Case start and end, Test Case journal entries (with test results), and the Invocable Component names used within a Test Case.

B.5.2 Result Codes

Essential

The specification must allow for Test Suite-defined result codes to be added on to those specified in the base set (IEEE Std 1003.3-1991 result codes). In particular, the reporting tools, API, and Test Case Controller must process result codes outside the scope of the base set.

Ensure that result codes provided in the base specification are broad enough to satisfy common test requirements.

B.5.3 Journal File Format

Essential

The format must be well defined, so that third parties can develop tools that manipulate the data.

B.6 Style Guide Requirements

B.6.1 Test Case Structure

Test Cases must be built in such a way that the granularity of the execution selections will be appropriate for the specification being tested.

B.6.2 Dynamic Configuration

Test developers should be encouraged to use parameters that allow for dynamic re-configuration of their tests in a stand-alone execution environment.

B.6.3 System Resource Requirements

Test Developers should be warned against having their tests require too much system resource for a given test. Developers must ensure that the Test Case requirements do not exceed the resources available in likely test environments.

B.6.4 Evolution

The Style Guide must clearly document how the specification of the Test Case Controller and Tools will evolve. In particular, it must show how extensions may be migrated into the base technology.

B.6.5 Portability of Tests

Test Developers should be encouraged to develop tests that are maximally portable.

B.6.6 Result Codes

Test Developers must use the predefined result codes whenever possible.

B.6.7 Required System Resources

Test Suites must document their system resource requirements.

B.6.8 Required Interfaces as Macros

Interfaces that are defined as macros in an implementation must be detected, and the installation process should adjust the Test Scenarios, Invocable Component name list, and Test Cases accordingly.

B.6.9 File System Use

Test Cases must only operate on test files in the working directory provided by the Test Case Controller.

Global Test Suite files should not be written to, only read from.

Appendix C - Project Description

C.1 Summary

This Project Description details the joint working relationship between the Open Software Foundation (OSF), Unix International (UI) and X/Open in the area of verification technologies, the process and results of the co-operative effort.

Organisations such as OSF, UI and X/Open produce or guide industry specifications for software products. When computer vendors implement these specifications, they use tests to verify conformance of the software to the specifications. These tests have in the past included the System V Verification Suite (SVVS) from AT&T, BSD Validation from Perennial, PCTS from Mindcraft, NIST-PCTS from NIST and VSX from X/Open.

These suites all have different user interfaces and different calling conventions, so testing organisations need to learn multiple technologies for verification work. In addition, to extend any of these suites test writers need to learn incompatible calling conventions.

C.2 Project Management

The Project is managed by the following representatives of each company:

- John Morris from OSF
- Shane McCarron from UI
- James de Raeve from X/Open

C.3 Background - Organisational Needs

OSF, UI and X/Open (hereafter "the Project") plan to co-operate in verification technologies to avoid further duplication of effort in the industry. To that end, the three organisations have agreed to define and produce a common verification interface and driver technology which will facilitate test development and execution. The following paragraphs describe the requirements each organisation is satisfying by participation in this project.

C.3.1 UNIX International

UI members have identified a requirement that conformance tests be developed and supplied independently of the operating system.

UI recognises that the requirements met by products will always be greater than the scope of common specifications. Therefore, the scope for common test technology will always be less than the requirement for product testing.

UI, by charter, does not develop software, and will therefore not itself develop test technology.

C.3.2 Open Software Foundation

OSF needs to distribute test suites for the verification of OSF technologies.

OSF has developed validation suites for functional areas of the AES. To date, these include the User Environment and Operating System areas. The User Environment test cases are independent of any test framework. The Operating System test cases have been developed to operate with the current version of VSX - the verification suite from X/Open. Both sets of test cases will be migrated to the specified interfaces.

C.3.3 X/Open Company Ltd.

X/Open has 3 clear goals in the field of test technology:

1. To maximise the quality and coverage of the X/Open test systems in order to best support the X/Open branding program;
2. To co-operate with UI, OSF and others in the development of conformance testing technology thus making the best use of resources;
3. To adopt a distribution/commercialisation policy for the X/Open Verification Suite that seeks to promote the market for open systems at the expense of revenue, yet ensure that the true costs of maintenance are recovered.

C.4 Project Description

The Project began at a joint meeting between OSF, UI and X/Open in London on September 29, 1989. At that meeting, the three organisations set base expectations for their future verification needs. The group then met with NIST on October 24th to discuss co-operation between the three organisations and NIST. NIST is supportive of the efforts of the Project and will participate in future talks on verification co-operation. On January 24, 1990, the three organisations met in Washington to detail the work program and structure the project management.

After the basic needs of the project were established, all three organisations agreed that it would be valuable to invite interested members of the test software development community to discuss their requirements for a test driver. This meeting occurred on January 24, 1990 in Washington and represented government, development and conformance testing interests. The project management group believes that through careful study of these requirements, a reasonably comprehensive and effective test driver can be specified. The management group further believes that if this specification process is successful, the technology will evolve into a suitable framework for the test development community.

C.5 Technology Descriptions

The deliverables for the project include:

- Requirements
- Architectural Specification

- Functional Specifications
- Installation Procedure
- Test Case Controller
- Report Generator
- API Library
- Documentation
    - User’s Manual
    - Programmer’s Manual
    - Style Guide
    - Manual Pages

[Requirement 1.4]

Users execute tests via the Test Case Controller (TCC). This single interface is capable of building and executing all tests provided by suites installed in the defined environment. While building and executing test cases, the TCC records messages and results in a journal. This journal may then be processed by a report generator to construct customised reports.

In order to take maximum advantage of the facilities provided by the toolkit environment, test suites are written using the defined API. The API Library contains routines that may be linked into executable test cases to interact with the TCC. These routines include interfaces to the journal file, entry and exit procedures for TCC startup and shutdown, and general-purpose routines for consistent test execution.

The documentation includes a User’s Manual for operation of the system, a Programmer’s Manual to assist programmers in writing test cases and manual pages for the TCC as well as the library routines.

The Project will provide driver extensibility and flexibility so that test developers can work within the structures of the driver and still produce useful, effective tests. The Project will not address the structure, content, or scope of Test Cases. A Test Case can be arbitrarily complex or simple and still function under the Driver. The structure of a Test Case can be based on assertions, functions, modules, or any other appropriate abstraction. It is up to the test developer to choose the most useful structure for a given test. This will often, but not always, be based on requirements of various conformance specifications. Because of the overriding concern for flexibility on the part of the test developer, the Driver will impose a minimum amount of structure on Test Cases.

C.6 Responsibilities

OSF has agreed to develop the driver and tools based on the requirements document. This agreement is based on the assumption that existing technology will be adapted to the needs of the group rather than invented from scratch.

UI has agreed to produce the supporting documentation, including the User's Manual and Programmer's Manual.

X/Open has agreed to move VSX to the new scaffold, and to develop the API library.

C.7 Technology Ownership

The technology, which is derived from many sources including OSF, Unix International and X/Open, will be copyrighted and available on a royalty free basis to the industry. OSF, UI and X/Open will make the technology available to third parties. There will be no sub-licence for the technology, and derivative works are encouraged. Derivative works must retain the copyright notice contained in the distributed source, and the use of the product name selected by the project will only be granted to versions that conform to the specification agreed to by OSF, UI and X/Open. *[Requirement 1.2]*

C.8 Maintenance

Maintenance of supplied technologies will be provided by the respective development organisations. All changes to the scaffold will be made in agreement with UI and X/Open. There will be no formal defect reporting system; all defect reports will be received through each company's normal reporting mechanisms. Versions of the technology will be delivered to UI, OSF and X/Open members for review before distribution as a released technology.

CONTENTS