Customizing the Test Execution Flow

This topic explains how to define a custom test execution flow for cross-platform testing (e.g., for building tests using the cross compiler and then performing platform-specific operations for deployment, test execution, and result collection).

Section includes:

- Understanding Test Execution Flow
- Customizing the Test Execution Flow
- Defining a Custom Test Execution Flow: Advanced
- Defining a Test Flow with a Socket Communication Channel
- Selecting External Embedded Debugging Mode
- Changing the Test Entry Point Function
- Customizing the Test Execution Flow Definition - Example
- Sample FTP Automation Scripts
- Test Flow Descriptions and Examples

Understanding Test Execution Flow

When you run a Test Configuration set to execute tests, C++test performs a series of actions that usually lead to the unit testing results being loaded into the C++test UI. These actions are specified in a test flow definition, which is stored in XML format and saved as part of the Test Configuration. All built-in Test Configurations that perform test execution have corresponding preconfigured test flow definitions.

In addition to providing preconfigured execution flows designed specifically for host-based testing, C++test also allows you to customize test flows to provide an easy way of working with non-standard environments.

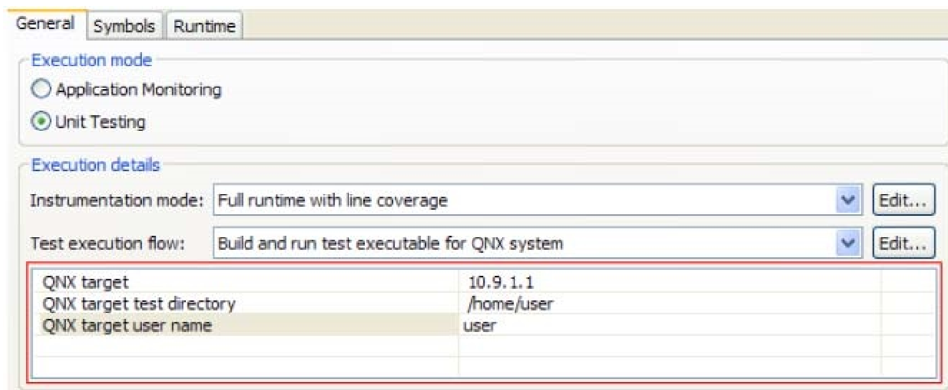
Customizing the Test Execution Flow

You can configure test execution flow parameters to suit your specific needs by using the editor available in the **Execution> General> Execution Details** section of the Test Configuration manager.

In most cases, there is no need to create a custom version of the execution flow because built-in Test Configurations allow easy editing of the most critical and commonly-modified flow properties.

To define a custom execution flow for a user-defined Test Configuration:

1. Choose **Parasoft> Test Configurations** to open the Test Configuration panel.
2. Open the **Execution> General** tab.
3. To modify existing properties, edit the parameters directly in the properties table.

   If you right-click in the table, the following commands are available:
   - **Restore default value**
   - **Insert file location**
   - **Insert directory location**
4. To adjust a test flow property that is not currently shown in the table (e.g. properties that you expect to modify frequently):
   a. Select the appropriate flow, then click **Edit**.
   b. Find that property definition in the test flow. It will look like:

   ```
   <SetProperty key="property_key"
   value="property_default_value"
   ```

   c. Add two additional attributes — `uiEditable="true"` and `displayName="user_friendly_name"` — so that the property definition looks as follows

```
<SetProperty key="property_key"
value="property_default_value"
uiEditable="true"
displayName="This is a customizable property" />
```

Since the `uiEditable="true"` attribute is added to the `SetProperty` flow step, this property will be customizable in the properties table. It will be listed here using the value of the `displayName` attribute. The value attribute will be used as a default value.

   d. Click **OK** to save the modified file. The XML document will be validated. If any problems are found during the validation, they will be reported.

5. Click **Apply**, then **OK** to save the modified Test Configuration.

The customizations will then be saved with the Test Configuration.

Defining a Custom Test Execution Flow: Advanced

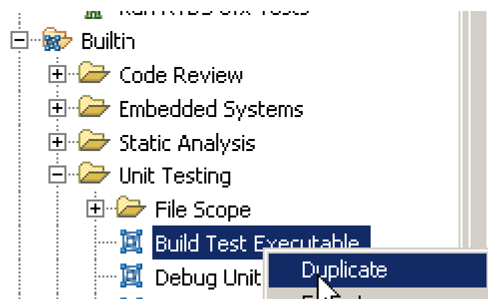
We recommend that you try to modify the properties list (as described above) first, then—if additional flow customization is required—you define a custom test flow.

Custom test execution flow definitions are stored in XML format and saved as part of the Test Configuration (which makes it possible to share it across the team). Typically, such definitions describe the sequence of steps necessary to perform unit testing in a non-standard environment. They can also be used to start any command line utility. In fact, a custom test flow definition can include C++test internal actions or external utilities started as processes in the operating system. The default test flow can be modified by:

- Removing existing steps
- Adding new steps
- Customizing the parameters of existing steps

To define a custom execution flow for a user-defined Test Configuration:

1. Choose **Parasoft> Test Configurations** to open the Test Configuration panel.
2. Expand the **Built-in> Unit Testing** folder.
3. Right-click **Build Test Executable**, then choose **Duplicate** from the shortcut menu.



4. Ensure that **User-defined> Build Test Executable** is selected.
5. Enter a new name for this Test Configuration (for example, something like `Build <target> Executable`).
6. Open the **Execution> Symbols** tab and clear the **Use stubs found in** check box.

> **Note**
>
> If you build the test executable without disabling the **Use stubs found in** option, you will get the following error when building the test executable:
>
> ```
> Error preprocessing file "C:\Program
> Files\ParaSoft\C++test9.0\plugins\com.parasoft.xtest.libs.cpp.win32.x86_9.0.4.43\
> os\win32\x86\etc\safestubs\safe_stubs_Win32.c":
>
> Process exited with code: 1
> ```

7. Edit the test flow:
   a. Open the **Execution> General** tab.
   b. Select **Custom flow (license required)** from the **Test Execution flow** combo box, then click the **Edit** button.
   c. Enter your modified test flow by choosing an existing test flow from the **Available built-in test execution flow** box, clicking **Restore**, then editing the XML for that test flow. See Customizing the Test Execution Flow Definition - Example for details on how to modify the flow definition.
8. Click **OK** to save the modified file. The XML document will be validated. If any problems are found during the validation, they will be reported.
9. Click **Apply**, then **OK** to save the modified Test Configuration.

Defining a Test Flow with a Socket Communication Channel

In embedded environments where TCP/IP sockets are available, it is common to use them to automate the test flow. Test execution using a socket communication channel typically involves:

1. Building the test executable by running the appropriate Test Configuration.
    - If necessary, modify this Test Configuration's test flow to suit your specific needs. In most cases, the only difference is the extension of the generated executable file and the settings of specific network-related values (ports, IP number etc.).

    Sample Test Flow Modification

    ```
    <SetProperty key="results_port" value="2567" />
    <SetProperty key="coverage_port" value="2568" />
    <TestRunnerWithSocketsGenerationStep
    testSuiteConfigFile="${cpptest:testware_loc}/testsuites.xml"
    testrunnerCFile="${cpptest:testware_loc}/cpptest_testrunner.c"
    testrunnerCppFile="${cpptest:testware_loc}/cpptest_testrunner.cpp"
    resultsHost="10.9.1.30"
    testLogPort=${cpptestproperty:results_port}
    covLogPort=${cpptestproperty:coverage_port}
    />
    <CompileStep />
    <LinkStep result="${cpptest:testware_loc}/${project_name}Test.foo"/>
    ```

    The first part (TestRunnerWithSocketsGenerationStep) prepares the test framework to run using specific network values (IP, ports – taken from property file).

    The second part compiles the test executable.

    The last part (LinkStep) is responsible for calling the linker in order to obtain the final test exec (called $ {project_name}Test.foo here; you will need to change the extension to match your system's specifics).

    Note that the TestRunnerWithSocketsGenerationStep should be used instead of the default TestRunnerGenerationStep.

2. Deploying the test executable to the target device.
    - *This step cannot always be automated.* If your system does not support automatic deployment, you might need to manually deploy the executable to the target device. Otherwise, you can use the FTP protocol, a synchronization tool, or any other means that your system provides.
3. Running the C++test SocketListener tool, which is configured to collect the results through a socket communication channel.
    - For details on how to perform this step (and step 4), see the following tip box.
    - After you run the program, it will start waiting for the results to arrive on the specified ports.
4. Starting the test executable. After initialization, it will open two ports and try to send data to C++test. This data will be collected by the listening agent and loaded into C++test for further analysis.
    - For details on how to perform this step (and step 3), see the following tip box.

Sample Test Flow Steps Covering Steps 3 and 4

```
<CustomStep
id="run_socket_listeners"
label="Running Socket Listeners..."
commandLine="&quot;java&quot; -cp &quot;${cpptest:cfg_dir}/../lib/source/socket_listener&quot;
SocketListener --channel &quot;${cpptestproperty:results_port}@$
{cpptest:testware_loc}/cpptest_results.tlog&quot; --channel
&quot;${cpptestproperty:coverage_port}@$
{cpptest:testware_loc}/cpptest_results.clog&quot; -sf
&quot;${cpptest:testware_loc}/sync_file&quot; -to 60"
workingDir="${cpptest:testware_loc}"
result="${cpptest:testware_loc}/cpptest_results.res"
runInBackground="true"
/>
<CustomStep
id="run_synchronization"
label="Running Synchronization..."
commandLine="&quot;java&quot; -cp &quot;${cpptest:cfg_dir}/../lib/source/socket_listener&quot;
Synchronize -sf &quot;${cpptest:testware_loc}/sync_file.init&quot; -to 60"
workingDir="${cpptest:testware_loc}"
result="${cpptest:testware_loc}/cpptest_results.res"
runInBackground="false"
/>
<CustomStep
id="run_test_exec"
label="Running Tests..."
commandLine="${cpptest:testware_loc}/${project_name}Test.foo" workingDir="${cpptest:testware_loc}"
result="${cpptest:testware_loc}/cpptest_results.res"
runInBackground="false"
/>
<CustomStep
id="run_synchronization"
label="Running Synchronization..."
commandLine="&quot;java&quot; -cp &quot;${cpptest:cfg_dir}/../lib/source/socket_listener&quot;
Synchronize -sf &quot;${cpptest:testware_loc}/sync_file.final&quot; -to 60"
workingDir="${cpptest:testware_loc}"
result="${cpptest:testware_loc}/cpptest_results.res"
runInBackground="false"
/>
```

The above steps perform the following actions (in the listed order):

- Run the SocketListener Java class, which is configured to listen on predefined ports. It also uses the -sf switch to create a synchronization file (in this case, sync_file).
- Run the Synchronize Java class. Note that the first step has the runInBackground="true" attribute; this means that the next step can run without waiting for that one to complete. This is necessary because SocketListener does not complete until the listening stops. When SocketListener initializes and is ready to receive results, it creates the synchronization file (sync_file.init, in this case) so that the run_synchronization step can complete and test_exec can be run. This prevents test execution from starting without proper listening agent initialization, which would cause the test results to be lost.
- The run_test_exec step is then started, and the test executable is run. It connects to SocketListener and transmits results over the TCP/IP protocol.
- After the transmission ends, SocketListener creates the .tlog and .clog files (test and coverage information files). Another synchronization file (sync_file.final) tells the synchronization step that everything is complete and results can be read.

Selecting External Embedded Debugging Mode

Selecting External Embedded debugging mode can only be done by direct modification to the Test Flow recipe. Insert the following line into the Test Flow recipe somewhere near the start of the first <RunnableExecution> section; among other unpublished properties:

```
<SetProperty key="emb_dbg" value="true" />
```

For examples, please review the built-in Test Configurations for environments for which the External Embedded mode is supported (see appropriate chapters).

For information about External Embedded debugging mode see Debugging Test Cases.

Changing the Test Entry Point Function

Entry point macros control the testing entry point. By default, main() is called. However, sometimes the tests can only be called from a different point in the program you are testing, or need to be called in a special manner. You can control this using the following macros:

Name	Description
CPPTEST_EPT_main	If defined, the 'int main(int, char*[])' is used as an entry point.
CPPTEST_EPT_wmain	If defined, the 'int wmain(int, wchar_t*[])' is used as an entry point.
CPPTEST_EPT_WinMain	If defined, the 'int WINAPI _tWinMain(HINSTANCE, HINSTANCE, LPSTR, int)' is used as an entry point.
CPPTEST_EPT_WinMain	If defined, the 'int WINAPI _tWinMain(HINSTANCE, HINSTANCE, LPWSTR, int)' is used as an entry point.
CPPTEST_EPT_void_main	If defined, the 'void main(int, char*[])' is used as an entry point.
CPPTEST_EPT_main_no_args	If defined, the 'int main(void)' is used as an entry point.
CPPTEST_EPT_void_main_no_args	If defined, the 'void main(void)' is used as an entry point.
CPPTEST_ENTRY_POINT_C_LINKAGE	If defined, the main function declaration starts with 'extern "C"' if compiled as c++ code.

You can also define the CPPTEST_ENTRY_POINT macro. If this macro will be defined, the generated main function will look like this:

```
    CPPTEST_ENTRY_POINT_RETURN_TYPE
CPPTEST_ENTRY_POINT(CPPTEST_ENTRY_POINT_ARGS)
{
        CppTest_Main(CPPTEST_ENTRY_POINT_ARGC, CPPTEST_ENTRY_POINT_ARGV);
        CPPTEST_ENTRY_POINT_RETURN_STATEMENT
}
```

CPPTEST_ENTRY_POINT_RETURN_TYPE, CPPTEST_ENTRY_POINT_ARGS, CPPTEST_ENTRY_POINT_ARGC, CPPTEST_ENTRY_POINT_ARGV and CPPTEST_ENTRY_POINT_RETURN_STATEMENT have the following default values that can be redefined:

Name	Value
CPPTEST_ENTRY_POINT_RETURN_TYPE	int
CPPTEST_ENTRY_POINT_ARGS	void
CPPTEST_ENTRY_POINT_ARGC	0
CPPTEST_ENTRY_POINT_ARGV	0
CPPTEST_ENTRY_POINT_RETURN_STATEMENT	return 0;

You can also define the CPPTEST_ENTRY_POINT_DEFINED macro. Defining this macro prevents the main routine from generating. In these cases, you need to call a `CppTest_Main(0, 0)` function to execute the test cases.

You must include the `cpptest.h` header file in the source file that contains the `CppTest_Main(0, 0)` call. Do not call the `CppTest_Main(0, 0)` from a function that is called during unit testing. Doing so will result in a loop where `CppTest_Main(0, 0)` is called infinitely.

Be sure to control changes to the source file with the PARASOFT_CPPTEST macro. For example:

```
#ifdef PARASOFT_CPPTEST
#include "cpptest.h"
#endif
...
#ifdef PARASOFT_CPPTEST
    CppTest_Main(0, 0);
#endif
```

These macros can be set in the Compiler Options area of the Project Options' Build Settings.

See Setting Project and File Options for details on project options.

Customizing the Test Execution Flow Definition - Example

Here is a sample test flow for Windows host-based unit testing:

```
<?xml version="1.0" encoding="UTF-8"?>
<FlowRecipeTemplate toolName="C++test" formatVersion="1.0">
        <Name>Default host-based unit testing</Name>

        <RunnableExecution>

        <SetProperty key="stub_config_file" value="${cpptest:testware_loc}/stubconfig.xml" />
        <SetProperty key="stub_config_header_file" value="${cpptest:testware_loc}/cpptest_stubconfig.h" />

        <TestCaseFindingStep
                testSuiteConfigFile="${cpptest:testware_loc}/testsuites.xml" allowNoTestCasesRun="false"
        />

        <PrecompileStep />
                <AppendIncludedTestCases />
                <HarnessInstrumentationStep />
                <ReadStaticCoverageStep />
                <SendStaticCoverageStep />
                <UserStubsInstrumentationStep />
                <ReadSymbolsDataStep />
                <LsiStep />
                <ReadLsiConfigStep />

        <AnalyzeMissingDefinitions stopOnUndefined="true" generateS-tubs="false" />

        <ConfigureStubs />

        <CreateStubConfigHeader />

        <TestRunnerGenerationStep
                testSuiteConfigFile="${cpptest:testware_loc}/testsuites.xml"
                testrunnerCFile="${cpptest:testware_loc}/cpptest_testrunner.c"
                testrunnerCppFile="${cpptest:testware_loc}/cpptest_testrunner.cpp"
                testLogFile="${cpptest:testware_loc}/cpptest_results.tlog"
                covLogFile="${cpptest:testware_loc}/cpptest_results.clog"
        />

        <CompileStep />

        <LinkStep result="${cpptest:testware_loc}/${project_name}Test.exe"/>

        </RunnableExecution>
```

```
            <ExecuteTestsExecution>
                       <RemoveFileStep
                               file="${cpptest:testware_loc}/cpptest_results.tlog"
                       />

        <CustomStep
                id="run_tests"
                label="Running tests..."
                commandLine="&quot;${cpptest:testware_loc}/${project_name}Test.exe&quot; --start-
after=${cpptestprop-erty:test_case_start_number}"
                workingDir="${cpptest:test_exec_work_dir}"
                result="${cpptest:testware_loc}/cpptest_results.tlog"
                timeoutTrackFile="${cpptest:testware_loc}/cpptest_results.tlog"
                timeoutInfoProperty="test_exec_timeouted"
                runInDebugger="${cpptest:run_in_debugger}"
        />

        <ReadTestLogStep
                testLogFile="${cpptest:testware_loc}/cpptest_results.tlog" timeoutInfoProperty="
test_exec_timeouted"
        />

        <ReadDynamicCoverageStep
                covLogFile="${cpptest:testware_loc}/cpptest_results.clog"
        />

           </ExecuteTestsExecution>

                   <RunnableExecution>
                                    <ClearTempCoverageData />
                                                        </RunnableExecution>
</FlowRecipeTemplate>
```

Note

- runInBackground specifies whether the step should be run in the background or foreground. When set to `true`, it allows the following action to run immediately, without waiting for the previous action to finish. When set to `false`, the running process must finish before the next-in-order can be started. The default value is `false`.

If the custom (cross) compiler definition was added correctly and the prebuilt host C++test runtime library was replaced with the cross-compiled target-specific one (see Configuring Testing with the Cross Compiler), then C++test should be able to successfully perform almost all actions— except the steps used for starting the test executable and loading the test log files. These steps are described below.

```
<CustomStep
id="run_tests"
label="Running tests..."
commandLine="&quot;${cpptest:testware_loc}/${project_name}Test.exe&quot;" workingDir="${cpptest:testware_loc}"
result="${cpptest:testware_loc}/cpptest_results.tlog"
runInBackground="false"
timeoutTrackFile="${cpptest:testware_loc}/cpptest_results.tlog"
/>
<ReadTestLogStep
testLogFile="${cpptest:testware_loc}/cpptest_results.tlog"
/>
<ReadDynamicCoverageStep
covLogFile="${cpptest:testware_loc}/cpptest_results.clog"
contextID=""
/>
```

You will probably need to replace these steps with ones suitable for the embedded environment you are working with. For example, assume your target is an Embedded Linux based Power PC board that supports FTP but does not support remote execution (no rsh or any similar utilities). In this case, the general schema for testing could be:

1. Prepare the test executable and perform the cross build (this should be handled by the default set of actions but with the cross compiler).
2. Deploy the test executable to the target board using FTP.
3. Wait for test execution to complete (the test executable will be started by the agent located on the target).
4. Download the test results using FTP.

For instance, let’s consider a trivial implementation of this schema.

If this is the existing step:

```
<CustomStep
id="run_tests"
label="Running tests..."
commandLine="&quot;${cpptest:testware_loc}/${project_name}Test&quot;" workingDir="${cpptest:testware_loc}"
result="${cpptest:testware_loc}/cpptest_results.tlog" runInBackground="false"
timeoutTrackFile="${cpptest:testware_loc}/cpptest_results.tlog"
/>
```

You could have this:

```
<CustomStep
id="deploy_test"
label="Deploying tests..."
commandLine="&quot;/home/machine/user/cptests/helper/Store.py${project_name}Test&quot;"
workingDir="${cpptest:testware_loc}"
/>
<CustomStep
Customizing the Test Flow 341
id="sleep"
label="Waiting for test execution..."
commandLine="sleep 30"
workingDir="${cpptest:testware_loc}"
result="cpptest_results.tlog"
runInBackground="false"
/>
```

- The FTP transfer was automated using the helper python script, which is provided at the end of this topic.

When test execution flow reaches the “sleep” step, the test executable should already be deployed to the target board. The sleep command is necessary to provide trivial synchronization so that the next step (downloading test results) will not start before test execution completes.

In production environments, you would typically want to implement more accurate synchronization (for example, with the help of a file marker being created after the test execution is finished). In this example, we will simply use sleep, which stops the flow execution before trying to access the results files on the target platform. It requires a simple agent on the target platform—an agent that waits for the uploaded test executable and starts it. A simple example of such a script is presented below:

```
#!/bin/bash
while [ 1 ]
do
if [ -e *Test ]; then
# Give it some time to finish upload...
# test executable transfer may be in progress
echo "Found test executable, waiting for upload to be finished..." sleep 10
echo "Cleaning up before tests..."
rm results.*log
echo "Adding exe perm .... to *Test"
chmod +x *Test
# execute
echo "Executing *Test"
./*Test
# Remove test executable
echo "Removing *Test"
rm -f ./*Test
else
echo "Test executable not detected ..." fi
echo "Sleeping..."
sleep 3
done
```

Again, it does not implement any synchronization; it just looks for a specified pattern of file names to appear in file system. Once it appears, it waits some extra time for the upload to finish (this can be replaced by creating a file marker on the host side after the upload is finished) and then the test executable is started, and results files (results.tlog with test results and results.clog with coverage results) are created. These files need to be downloaded back to the host machine and loaded into C++test. To accomplish this, you need to modify another piece of the test flow definition. Before these "reading log" steps...

```
<ReadTestLogStep
testLogFile="${cpptest:testware_loc}/cpptest_results.tlog" />
<ReadDynamicCoverageStep
covLogFile="${cpptest:testware_loc}/cpptest_results.clog" contextID=""
/>
```

you would add custom steps for downloading the results files from the target platform; for example:

```
<CustomStep
id="results_dwl"
label="downloading tests results..."
commandLine="&quot;/home/machine/user/cptests/helper/Get.py cpptest_results.tlog&quot;"
workingDir="${cpptest:testware_loc}"
result="cpptest_results.tlog"
runInBackground="false"
/>
<CustomStep
id="coverage_dwl"
label="downloading coverage results..."
commandLine="&quot;/home/machine/user/cptests/helper/Get.py cpptest_results.clog&quot;"
workingDir="${cpptest:testware_loc}"
result="cpptest_results.clog"
runInBackground="false"
/>
```

Having this set up, you should be able to automatically perform the complete testing loop with unit test execution on the target device. Depending on the development environment, numerous variations of this example can be applied.

Sample FTP Automation Scripts

This section provides sample Python scripts for automating FTP operations.

Store.py

```
#!/usr/bin/python

import sys
print "Argv = ", sys.argv
from ftplib import FTP
ftp = FTP('10.9.1.36', 'user', 'pass') ftp.set_debuglevel(1)
print ftp.getwelcome()
file = open(sys.argv[1], "rb")
try:
             ftp.storbinary("STOR %s" % sys.argv[1], file)
finally:
             file.close()
                ftp.quit()
```

Get.py

```
#!/usr/bin/python

import sys
print "Argv = ", sys.argv
if len(sys.argv) < 2:
     print "Too few arguments"
from ftplib import FTP
ftp = FTP('10.9.1.36', 'user', 'pass') ftp.set_debuglevel(1)
print ftp.getwelcome()
file = open(sys.argv[1], "wb")
try:
     ftp.retrbinary("RETR %s" % sys.argv[1], file.write)
finally:
     file.close()
ftp.quit()
```

Test Flow Descriptions and Examples

Execution Control Elements

These are highest level elements (besides the 'FlowRecipeTemplate' document root) of every recipe. They are containers for execution flow steps (commands): steps have to be placed inside their context and are executed sequentially in it. However, they cannot be nested. Introducing a new one requires closing its predecessor.

RunnableExecution

Unconditionally execute contained commands.

Example:

```
<RunnableExecution>
    <Echo msg="Hello world!" />
</RunnableExecution>"
```

ConditionalExecution

Conditionally execute contained commands. Useful when test value contains one of C++test execution flow variables described later.

Attributes:

- value - The test value (value to test).
- equals - The comparison value that test value must match. Cannot co-exist with 'notEquals'.
- notEquals - The comparison value that test value mustn't match. Cannot co-exist with 'equals'.

Example:

```
 <ConditionalExecution value="${cpptest:os}" equals="unix">
     <Echo msg="This is UNIX-like OS." />
</ConditionalExecution>"
```

ExecuteTestsExecution

Execute contained commands in a loop until all Test Cases have been executed. This element is only useful/valid when contains the test execution step (a `'CustomStep'` launching the Test Executable) followed by the `'ReadTestLogStep'`. Inside the `{cpptestproperty:test_case_start_number}` variable is updated automatically and shall be passed to the `'--start-after='` parameter of the Test Executable.

Example:

Example

```
<ExecuteTestsExecution>
    <CustomStep commandLine="&quot;${cpptest:testware_loc}/
        ${project_name}Test.exe&quot; --start-after=${cpptestprop-erty:test_case_start_number}"
...other attributes... />
    <ReadTestLogStep testLogFile="${cpptest:testware_loc}/cpptest_results.tlog" />
</ExecuteTestsExecution>"
```

Commands

AnalyzeMissingDefinitions

Internal step used to analyze information about missing symbols. It can be also used to automatically generate stubs for missing symbols if the appropriate attribute is set. If (after optional stub generation) there are still some missing symbols, execution flow will be stopped if the Execution -> Symbols -> Perform early check for potential linker problems option is checked in the Test Configuration.

Attributes:

- generateStubs -- If defined to "true" auto stubs will be generated for missing symbols. By default it is set to "false".
- generateUserStubs -- If defined to "true", the stub wizard will be opened to enable definition of user stubs for missing symbols. By default it is set to "false".

Example:

```
<AnalyzeMissingDefinitions generateStubs="true" />
```

AppendIncludedTestCases

Internal step that bundles the included test suite files with the appropriate project source files.

Example:

```
<AppendIncludedTestCases />
```

ClearTempCoverageData

Internal step used to clear temporary static and dynamic coverage data. It should be included at the end of the execution flow if there is no need to store static coverage data for future use. If you have two execution flow recipes that are meant to be run in sequence (for example, "Build test executable" and "Read test results"), only the second one should contain ClearTempCoverageData step at the end.

Example:

```
<ClearTempCoverageData />
```

CompileStep

Internal step that compiles the instrumented source files to be linked into the test executable.

Example:

```
<CompileStep />
```

ConditionalStop

Stops test execution flow if given the condition is fulfilled or not fulfilled (depending on attributes).

Deprecated - This command will be removed in subsequent versions of C++test. Use combinations of 'ConditionalExecution' control element and unconditional 'Stop' instead.

Attributes:

- property -- Name of the execution flow property to check.
- equals -- Execution is stopped when the property has the specified value. This attribute is optional.
- notEquals -- Execution is stopped when property has value a value other than the one specified. This attribute is optional.

Example:

```
<ConditionalStop property="test_exec_timeouted" equals="true" />
```

ConfigureStubs

Internal step used to create stub configuration xml file. The location of the resulting file is controlled by the "stub_config_file" execution flow property.

Example:

```
<ConfigureStubs />
```

CustomStep

Runs external executable.

Attributes:

- commandLine -- Defines command line to be executed.
- workingDir -- Defines working directory for created process.
- runInBackground -- If set to "true" process is run asynchronously. This attribute is optional. By default it is set to "false".
- runInDebugger -- If set to "true" process is run the debugger as configured in the Test Configuration. This attribute is optional. By default it is set to "false".
- okExitCode -- If specified, the exit code of the process will be checked and test execution flow will be stopped in case of failure. This attribute is optional.
- stdIn -- If specified, contents of the provided file will be used as the stdin for the process. This attribute is optional.
- stdOut -- If specified, stdout from the process will be written to the provided file.This attribute is optional.
- stdErr -- If specified, stdout from the process will be written to the provided file. This attribute is optional.
- responseFileSupported -- Tells whether the process supports reading command line parameters from the response file specified with @file. In this case, when the process command line exceeds the maximum length (specified with the com.parasoft.xtest.common.process.maxcmdlength Java parameter; 32000 by default), command line parameters will be passed to the process via a response file. This attribute is optional. By default it is set to "false".
- dependencyFiles -- Defines the process dependency files. It is used along with the "result" attribute to determine whether a step can be skipped as "up-to-date" when the command line did not change since the previous run and result file is newer than all of the dependency files. This attribute is optional.
- result -- Defines the process result file. It is used along with the "dependencyFiles" attribute to determine whether a step can be skipped as "up-to-date" when the command line did not change since the previous run and result file is newer than all of the dependency files. This attribute is optional.
- id -- Specifies the step's identifier.
- label -- Specifies the step's label (to be printed in the C++test console).
- bypassConsole -- If set to "true", the output from the process will not be printed on the C++test console. This attribute is optional. By default it is set to "false".
- timeoutTrackFile -- If specified, C++test will use timeout specified in test configuration for the run executable. The specified file will be checked periodically and if it's not modified within a time period longer than specified timeout, the process will be stopped. This attribute is optional.
- timeoutInfoProperty -- Specifies the name of the test unit property to be set to "true" when timeout occurred. This attribute is optional.

Example:

```
<CustomStep
        id="ctdt_nm"
        label="CTDT Generation - extracting symbols..."
        commandLine="&quot;${cpptestproperty:nm}&quot; -g ${cpptest:test_objects_quoted}"
        workingDir="${cpptest:testware_loc}"
        stdOut="${cpptest:testware_loc}/ctdt_nm.out"
        result="${cpptest:testware_loc}/ctdt_nm.out"
        bypassConsole="true"
        dependencyFiles="${cpptest:test_objects_quoted}"
/>
```

CreateStubConfigHeader

Internal step used to produce a stub configuration header file to be included by the instrumented sources. The location of the resulting header is controlled by the "stub_config_header_file" execution flow property.

Example:

```
<CreateStubConfigHeader />
```

Echo

Prints given message on the C++test console or to the file.

Attributes:

- msg -- Defines the message to be printed.
- label -- Defines the label of this flow step to be printed on the C++test console.
- file -- If specified, C++test will print the given message to the file specified (rather than on the console). This attribute is optional.

Example:

```
<Echo msg="Hello world!" />
```

HarnessInstrumentationStep

Internal step instruments the project source files.

Example:

```
<HarnessInstrumentationStep />
```

LinkStep

Links the test executable using previously prepared object files.

Attributes:

- result -- Specifies the location of the output test executable files.
- linker -- If specified, defines the linker executable to be used instead of the one defined in the project properties. This attribute is optional.
- linkerCmdLine -- If specified, defines the linker command line pattern to be used instead of the one defined in the compiler definition file. This attribute is optional.

Example:

```
<LinkStep result="${cpptest:testware_loc}/${project_name}Test.exe" />
```

LsiStep

Internal step used to analyze information about used and available symbols.

Attributes:

- libSymFile - Specifies the file with external symbols list to read. Optional.

  See Testing VxWorks DKM Projects, Providing an External List of Library Symbols, and Generating an External VxWorks Image Symbols List.

Example:

```
<LsiStep />
```

PrecompileStep

Internal step that precompiles the project source files for the LSI analysis.

Example:

```
<PrecompileStep />
```

PrepareDataSources

Converts managed data sources into the format supported by C++test test execution runtime library— either a CSV file or a source file containing a source code array to be compiled into test executable.

Attributes:

- generationDir -- Output directory for generated CSV or array source file. This attribute is optional. By default it is set to the C++test testware directory for current test unit.
- inputDir -- Location of the generated CSV file to be used when reading data source during the actual test case execution. It may differ from the "generationDir" if the actual test execution is done on a target device and the generated CSV file will need to be moved there. This attribute is optional. By default it is set to the value of the "generationDir" attribute.
- limit -- Specifies the maximum number of data source rows to be converted. A value less than zero means that all data source rows should be converted. This attribute is optional. By default it is set to -1.
- type -- Defines whether data should be converted to a CSV file or a source code array. Should be defined as "csv" or "array". This attribute is optional. By default it is set to "csv".

Example:

```
<PrepareDataSources limit="100" type="csv" />
```

ReadDynamicCoverageStep

Reads the log file containing coverage results from the test execution.

Attributes:

- covLogFile -- File to be read. Can contain '*' and '?' wildcards to read multiple files at once. To read the series of files obtained by using split file communication channel (see File Communication Channel Implementation), specify the path to the first file (e.g., 'cpptest_results.clog') and make sure that all the files are placed in the same directory and that their names follow the original naming scheme ('cpptest_results.clog', 'cpptest_results.clog.0001', 'cpptest_results.clog.0002', etc.). The rest of the files in the series will be automatically merged with the first one and then removed.

ReadLsiConfigStep

Internal step used to read data from the LSI module.

Example:

```
<ReadLsiConfigStep />
```

ReadNativeCoverage

Reads test coverage information from the native test cases imported from the C++test 2.3/6.x (if there are any in the current test unit).

Example:

```
<ReadNativeCoverage />
```

ReadNativeTestsLog

Reads test execution log from the native test cases imported from the C++test 2.3/6.x (if there are any in the current test unit).

Example:

```
<ReadNativeTestsLog />
```

ReadStaticCoverageStep

Internal step that reads the static coverage information file prepared when instrumenting project source files.

Example:

```
<ReadStaticCoverageStep />
```

ReadSymbolsDataStep

Internal step used to read information about symbols used / defined in processed source files.

Attributes:

- readStubsSymbols -- Defines whether information from processed stub files should be read. By default it is set to "true".

Example:

```
<ReadSymbolsDataStep />
```

ReadTestLogStep

Reads test execution results from the test log.

Attributes:

- testLogFile -- Specifies the test log file to be read. To read the series of files obtained by using split file communication channel (see File Communication Channel Implementation), specify the path to the first file (e.g., 'cpptest_results.tlog') and make sure that all the files are placed in the same directory and that their names follow the original naming scheme ('cpptest_results.tlog', 'cpptest_results.tlog.0001', 'cpptest_results.tlog.0002', etc.). The rest of the files in the series will be automatically merged with the first one and then removed.
- timeoutInfoProperty -- Internal attribute that defines the test flow property to be checked to find out whether test execution has been stopped by the driver because of the timeout. Should be set to "test_exec_timeouted".
- logTime -- Defines the format pattern used to display the time for each test log message on the console. In the format string, all letters (a-z and A-Z) are treated as patterns. Text must be quoted with a single quote ("'"). Two single quotes ("''") are used to print one single quote. The following patterns are available:

y   Year
M Month
d   Day
a   Am/pm marker
H Hour (0-23)
h   Hour (1-12)
m Minute
s   Second
S   Fraction of seconds
Z   RFC 822 time zone

The minimum number of digits can be set by repeating a pattern letter. Shorter numbers are zero padded. If the year pattern has two letters (yy), a two digit year is used. If the month pattern has three or more letters (MMM), text will be used; otherwise, numbers will be used. If there is a fraction of seconds pattern, it sets the exact number of digits used (precision). This attribute is optional.

Example:

```
<ReadTestLogStep
        testLogFile="${cpptest:testware_loc}/cpptest_results.tlog"
        timeoutInfoProperty="test_exec_timeouted"
        logTime="'Date': YY/MM/DD, 'Time': HH:mm:ss.SSSS"
/>
```

RemoveFileStep

Removes given file/files from the file system.

Attributes:

- file -- Defines file to be removed. Can contain '*' and '?' wildcards to remove multiple files at once.

Example:

```
<RemoveFileStep file="${cpptest:testware_loc}/*.clog" />
```

RunNativeTests

Executes the native test cases imported from the C++test 2.3/6.x (if there are any in the current test unit).

Example:

```
<RunNativeTests />
```

SendStaticCoverageStep

Internal step used to pass all read static coverage information to the coverage results system.

Example:

```
<SendStaticCoverageStep/>
```

SetProperty

Sets the execution flow property. You can use regular expressions with the 'search' and 'replace' attributes to search for and replace values . This is especially useful when the test value contains one of the C++test execution flow variables described later in this section.

Attributes:

- key -- Name of the property to be set.
- value -- Value of the property to be set.
- search -- Regular expression to search in value. Optional.
- replace -- Text to substitute for expression found. Required if the 'search' attribute is present.

Example 1:

```
<SetProperty key="stub_config_file" value="${cpptest:testware_loc}/stubconfig.xml" />
```

stores the plain value into `"${cpptestproperty:stub_config_file}"`

Example 2:

```
<SetProperty key="twl_mixed_path" value="${cpptest:testware_loc}" search="\\" replace="/" />
```

replaces all backslashes with forwardslashes for the provided value and stores the outcome into "${cpptestproperty:twl_mixed_path}":

Stop

Unconditionally stops the execution flow.

TestCaseFindingStep

Internal step used to prepare an xml file with the list of the test cases to be executed. The xml file is later used by the TestRunnerGenerationStep.

Attributes:

- testSuiteConfigFile -- Defines output xml file.
- allowNoTestCasesRun -- Specifies whether the execution flow should be stopped when there are no test cases to be executed.

Example:

```
<TestCaseFindingStep

        testSuiteConfigFile="${cpptest:testware_loc}/testsuites.xml" allowNoTestCasesRun="false"

/>
```

TestRunnerGenerationStep

TestRunnerWithSocketsGenerationStep

Prepares test runner source code that will work as a driver of the test case execution. Depending on the specified attributes, the test executable will produce log files or will send results via socket connection.

Attributes:

- testrunnerCFile -- Location of the test runner file (if the project contains only C code).
- testrunnerCppFile -- Location of the test runner file (if the project contains C++ code).
- testSuiteConfigFile -- The xml file with the list of test cases to be executed. This file should be prepared by the TestCaseFindingStep.
- testLogFile -- Defines where the test executable should produce the test execution log file. Can be skipped if socket communication is used.
- covLogFile -- Defines where the test executable should produce the test coverage log file. Can be skipped if socket communication is used.
- resultsHost -- Defines the host where the socket listener will wait for the test results.
- testLogPort -- Defines the port on which the socket listener will wait for the test execution results. Can be skipped if file communication is used.
- covLogPort -- Defines the port on which the socket listener will wait for the test coverage results. Can be skipped if file communication is used.
- appendLogs -- If file communication is used, defines whether the runtime should append new information to the test execution and coverage log files instead of overwriting them. By default it is set to "true".

UserStubsInstrumentationStep

Internal step that instruments the stub files.

Example:

```
<UserStubsInstrumentationStep />
```

Variables

Since the execution flow recipe is an xml document, all values used as flow step attributes need to be valid xml strings. This means that all occurrences of special characters (such as " or <) need to be substituted with appropriate escape sequences (for example, " or < respectively).

Variables that can be used in the flow step attributes are:

- ${workspace_loc} -- Workspace location.
- ${project_name} -- Name of the tested project.
- ${project_loc} -- Location of the tested project.
- ${project_loc:PROJECT_NAME} -- Location of the PROJECT_NAME project.
- ${resource_loc:RESOURCE_PATH} -- location of the resource represented by RESOURCE_PATH.
- ${cpptest:testware_loc} -- Location of the C++test testware for current test unit.
- ${cpptest:cfg_dir} -- Location of the C++test configuration files in the C++test installation directory.
- ${cpptest:utils_dir} -- Location of the C++test utilities in the C++test installation directory.
- ${cpptest:uid} -- Unique identifier.
- ${cpptest:add_file_ext} -- Extension of the additionally created source files, computed based on the extensions of the project source files.
- ${cpptest:os} -- The kind of OS: either "windows" or "unix" (linux).
- ${cpptest:run_in_debugger} -- "true" or "false" depending on the value of the "Run tests in debugger" option in Test Configuration.
- ${cpptest:test_exec_work_dir} -- Test executable working directory as specified in the Test Configuration.
- ${cpptest:test_object_files} -- Comma-separated list of object files that will be part of the test executable.
- ${cpptest:test_objects_quoted} -- Space-separated list of object files that will be part of the test executable; each file will be surrounded with quotes (").

- ${cpptest:test_objects_esc_quoted} -- Space-separated list of object files that will be part of the test executable; each file will be surrounded with escaped quotes (\").

FileSynchronizeStep

This step pauses the execution flow while waiting on a specific file to either be created or for the file to be inactive for a specified amount of time. Behavior depends on the attributes used described below.

Attributes:

- fileSync - the full path to the file that this step will attempt to sychronize with.
- timeout - length of time (in milliseconds) to wait for the file (specified with the fileSync attribute) to be created. If the file is not created by the end of the time-to-timeout, then the execution flow will resume as normal. This attribute takes precedence over fileInactiveTimeout attribute described below and cannot be used in conjunction with it.
- fileInactiveTimeout - the length of time (in milliseconds) to wait for file inactivity before timing out on the file specified through the fileSync attribute. This is done by continuously checking the time that the file was last modified. The execution flow will resume when C++test detects that the file is inactive for the specified length of time.

Examples:

The following setting will pause the execution flow for 10 seconds or until the `sync_file.txt` file appears at the specified location:

```
<FileSynchronizeStep
        fileSync="${cpptest:testware_loc}/sync_file.txt"
        timeout="10000"
/>
```

The following setting will track the timestamp of the `cpptest_results.tlog` file. When C++test detects that the file is inactive for 10 seconds, then the execution flow will resume:

```
<FileSynchronizeStep
        fileSync="${cpptest:testware_loc}/cpptest_results.tlog"
        fileInactiveTimeout="10000"
/>
```