MicroEJ Test-suite User Manual

Table of Contents

1. Definition 4
2. How to use the test-suite framework 5
    2.1. Using the MicroEJ Test-Suite engine 5
    2.2. Using the MicroEJ Test-Suite engine provided by a MicroEJ Platform 5
3. Using the MicroEJ Test-Suite Ant tasks 7
    3.1. The testsuite task 7
    3.2. The javaTestsuite task 8
    3.3. The htmlReport task 8
4. JUnit report generation 10
5. Example - Deploying an Ant test-suite 12
    5.1. Tests cases 12
    5.2. Creating the test harness 12
    5.3. Creating the MicroEJ Test-Suite launch 12
6. Example - Deploying a Java test-suite 15
    6.1. The MicroEJ environment 15
    6.2. Preview of the test-suite 15
    6.3. Creating the MicroEJ application launch 16
    6.4. The MicroEJ Test-Suite harness 18
    6.5. Creating the MicroEJ Test-Suite launch 19
7. Using the trace analyzer 22
    7.1. The TraceAnalyzer tasks options 22
    7.2. The FileTraceAnalyzer task options 22
    7.3. The SerialTraceAnalyzer task options 22
8. Appendix 23
    8.1. Specific custom properties 23

List of Figures

4.1. JUnit final report example 10
5.1. Example Ant - The tree files 12
5.2. Example Ant - Final JUnit report 13
5.3. Example Ant - Final HTML report 14
6.1. Example Java - Available platforms 15
6.2. Example Java - The tree files 16
6.3. Example Java - Main Tab 17
6.4. Example Java - Execution Tab 17
6.5. Example Java - Common Tab 18
6.6. Example Java - Final JUnit report 20
6.7. Example Java - Final HTML report 21

1 Definition

The MicroEJ Test-Suite is an engine made for validating any development project using automatic testing. The MicroEJ Test-Suite engine allows the user to test any kind of projects within the configuration of a generic ant file.

2 How to use the test-suite framework

The MicroEJ Test-Suite framework can be retrieved and used either from the JAR archive file providing only the MicroEJ Test-Suite engine, either from a MicroEJ Platform..

2.1 Using the MicroEJ Test-Suite engine

To use the MicroEJ Test-Suite framework from the archive JAR file, some dependencies used by the framework must be retrieved:

- asm library (4.0.0 or above): `asm-[version].jar`
- apache commons-lang (2.4.0 or above): `commons-lang-[version].jar`

The MicroEJ Test-Suite engine is an antlib and provide a xml file to describe all the Ant tasks available to the user. To use the antlib definition, here is an example on how to do so:

```
<project name="testsuiteDefinition" xmlns:testsuite="antlib:com.is2t.testsuite.ant">
 <target name="testsuite/definition">
  <path id="testsuite-engine.path">
   <fileset dir="${platform.dir}/tools">
    <include name="testsuite-engine*.jar"/>
    <include name="commons-lang*.jar"/>
    <include name="asm*.jar"/>
   </fileset>
  </path>
  <taskdef uri="antlib:com.is2t.testsuite.ant" resource="com/is2t/testsuite/ant/
antlib.xml" classpathref="testsuite-engine.path"/>
 </target>
</project>
```

By defining the antlib as explained with the namespace `testsuite`, the following tasks can be called from Ant:

- testsuite:testsuite
- testsuite:javaTestsuite
- testsuite:htmlReport

These tasks are defined and explained on the section Using the test-suite Ant tasks.

2.2 Using the MicroEJ Test-Suite engine provided by a MicroEJ Platform

The MicroEJ Platform containing the MicroEJ Test-Suite framework is provided with all the externals tools available to use the engine. To use the MicroEJ Test-Suite engine, import the file in `${platform.dir}/scripts/testsuiteDefinition.xml` and call the target `testsuite/definition`. This target will initialize the antlib and define all the available tasks from the MicroEJ Test-Suite engine.

Define the namespace `testsuite` in the Ant project: `xmlns:testsuite="antlib:com.is2t.testsuite.ant"`.

After the initialization, the following tasks can be called from Ant:

- testsuite:testsuite
- testsuite:javaTestsuite
- testsuite:htmlReport

These tasks are defined and explained on the section Using the test-suite Ant tasks.

The MicroEJ Platform will also provide some usefull components, such as the `Check Helper` Java library which can be used to make tests and the `Trace Analyzer` which can be used on the MicroEJ Test-Suite harness to analyze the results of any test from any output, file, serial, telnet…

3 Using the MicroEJ Test-Suite Ant tasks

Multiple Ant tasks are available in the testsuite-engine provided jar:

- `testsuite` allows the user to run a given testsuite and to retrieve an XML report document in a JUnit format.
- `javaTestsuite` is a subtask of the `testsuite` task, used to run a specialized testsuite for Java (will only run Java classes).
- `htmlReport` is a task which will generate an HTML report from a list of JUnit report files.

3.1 The testsuite task

This task have some mandatory attributes to fill:

- `outputDir`: the output folder of the test-suite. The final report will be generated at `[outputDir]/[label]/[reportName].xml`, see the `testsuiteReportFileProperty` and `testsuiteReportDirProperty` attributes.
- `harnessScript`: the harness script must be an Ant script and it is the script which will be called for each test by the test-suite engine. It is called with a `basedir` located at output location of the current test. The test-suite engine will provide to it some properties giving all the informations to start the test:
  - `testsuite.test.name`: The output name of the current test in the report. Default value is the relative path of the test. It can be manually set by the user. More details on the output name are available in the section Specific custom properties.
  - `testsuite.test.path`: The current test absolute path in the filesystem.
  - `testsuite.test.properties`: The absolute path to the custom properties of the current test (see the property `customPropertiesExtension`)
  - `testsuite.common.properties`: The absolute path to the common properties of all the tests (see the property `commonProperties`)
  - `testsuite.report.dir`: The absolute path to the directory of the final report.

Some attributes are optional, and if not set by the user, a default value will be attributed.

- `timeOut`: the time in seconds before any test is considerated as unknown. Set it to 0 to disable the time-out. Will be defaulted as 60.
- `verboseLevel`: the required level to output messages from the test-suite. Can be one of those values: error, warning, info, verbose, debug. Will be defaulted as info.
- `reportName`: the final report name (without extension). Default value is `testsuite-report`.
- `customPropertiesExtension`: the extension of the custom properties for each test. For instance, if it is set to `.options`, a test named `xxx/Test1.class` will be associated with `xxx/Test1.options`. If a file exists for a test, the property `testsuite.test.properties` is set with its absolute path and given to the `harnessScript`. By default, custom properties extension is `.properties`.
- `commonProperties`: the properties to apply to every test of the test-suite. Those options might be overridden by the custom properties of each test. If this option is set and the file exists, the property `testsuite.common.properties` is set to the absolute path to this file to the `harnessScript`. By default, there is not any common properties.
- `label`: the build label. Will be generated as a timestamp by the test-suite if not set.

- `productName`: the name of the current tested product. Default value is `TestSuite`.
- `jvm`: the location of your Java VM to start the testsuite (the `harnessScript` is called as is: `[jvm] [...] -buildfile [harnessScript]`). Will be defaulted as your `java.home` location if the property is set, or to `java`.
- `jvmargs`: the arguments to pass to the Java VM started for each test.
- `testsuiteReportFileProperty`: the name of the Ant property in which is stored the path of the final report. Default value is `testsuite.report.file` and path is `[outputDir]/[label]/[reportName].xml`
- `testsuiteReportDirProperty`: the name of the Ant property in which is store the path of the directory of the final report. Default value is `testsuite.report.dir` and path is `[outputDir]/[label]`
- `testsuiteResultProperty`: the name of the Ant property in which you want to have the result of the test-suite (true or false), depending if every tests successfully passed the test-suite or not. Ignored tests do not affect this result.

Finally, you have to give as nested element the path containing the tests.

- `testPath`: containing all the file of the tests which will be launched by the test-suite.
- `testIgnoredPath` (optional): Any test in the intersection between `testIgnoredPath` and `testPath` will be executed by the test-suite, but will not appear in the JUnit final report. It will still generate a JUnit report for each test, which will allow the HTML report to let them appears as "ignored" if it is generated. Mostly used for known bugs which are not considered as failure but still relevant enough to appears on the HTML report.

3.2 The javaTestsuite task

This task extends the `testsuite` task, specializing the test-suite to only start real Java class. This task will retrieve the classname of the tests from the classfile and will provide new properties to the harness script:

- `testsuite.test.class`: The classname of the current test. The value of the property `testsuite.test.name` is also set to the classname of the current test.
- `testsuite.test.classpath`: The classpath of the current test.

3.3 The htmlReport task

This task allow the user to transform a given path containing a sample of JUnit reports to an HTML detailled report. Here is the attributes to fill:

- A nested fileset containing all the JUnit reports of each test. Take care to exclude the final JUnit report generated by the testsuite.
- A nested element `report`
  - `format`: The format of the generated HTML report. Must be `noframes` or `frames`. When `noframes` format is choosen, a standalone HTML file is generated.
  - `todir`: The output folder of your HTML report.
  - The `report` tag accepts the nested tag `param` with `name` and `expression` attributes. These tags can pass XSL parameters to the stylesheet. The built-in stylesheets support the following parameters:
    - `PRODUCT`: the product name that is displayed in the title of the HTML report.

- `TITLE`: the comment that is displayed in the title of the HTML report.

`Tip:` It is advised to set the format to `noframes` if your testsuite is not a Java testsuite. If the format is set to `frames`, with a non-Java MicroEJ Test-Suite, the name of the links will not be relevant because of the non-existency of packages.

4 JUnit report generation

The test-suite engine will generate a final report in a junit format, which can be easily read in Eclipse via the JUnit view. This format allow the user to have a quick and easy review of the test-suite results.

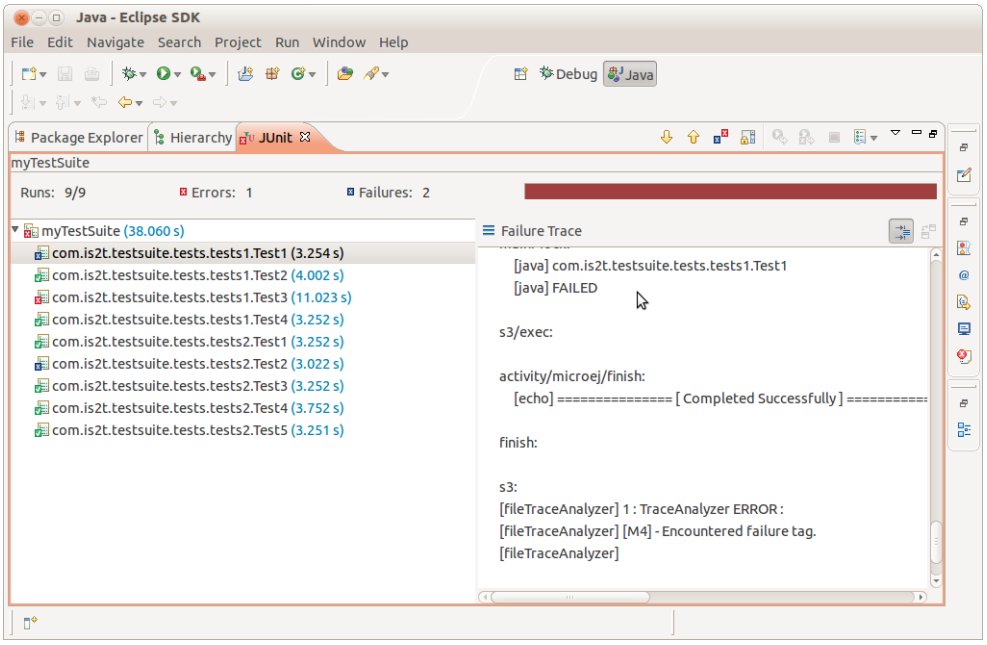


Figure 4.1. JUnit final report example

Every test might end in three different states.

- `Error`: When the `time out` is reached, and the test has not finished to run, the current test result will be considered as unknown, which will result as error in the JUnit report.
- `Failure`: When the `harness script` fails, the current test result will be considered as failure.
- `Passed`: When the `harness script` does not fail and ends within the given `time out`, the current test result will be considered as passed.

The execution trace of any test will be saved into the report, but the Junit view will only show the trace of a test which ended as an `error` or as a `failure`.

JUnit format is an XML format and the final report respects the following DTD:

```
<!ELEMENT testsuite (testcase+)>
<!ATTLIST testsuite
 errors CDATA #REQUIRED
 failures CDATA #REQUIRED
 hostname CDATA #REQUIRED
 ignored CDATA #REQUIRED
 name CDATA #REQUIRED
 started CDATA #REQUIRED
 tests CDATA #REQUIRED
 time CDATA #REQUIRED
 timestamp CDATA #REQUIRED
>

<!ELEMENT testcase ((failure|error|system-out|system-err)?)>
<!ATTLIST testcase
 classname CDATA #REQUIRED
 name CDATA #REQUIRED
 time CDATA #REQUIRED
>

<!ELEMENT failure (#PCDATA)>
<!ELEMENT error (#PCDATA)>
<!ELEMENT system-out (#PCDATA)>
<!ELEMENT system-err (#PCDATA)>
```

5 Example - Deploying an Ant test-suite

The goal of this section is to describe the steps to deploy a test suite environment on a simple application. For this example, we will start a simple Ant Test Suite.

5.1 Tests cases

For this example, we have two Ant tests, `Test1.xml` and `Test2.xml`.

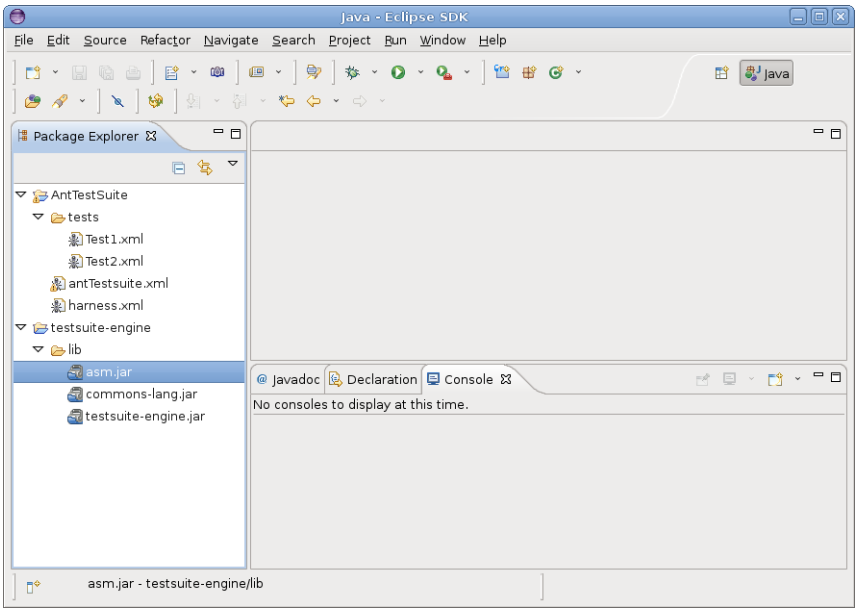


Figure 5.1. Example Ant - The tree files

1 Test1.xml

```
<project name="test1" default="build" >
 <target name="build" >
  <echo message="Test OK"/>
 </target>
</project>
```

2 Test2.xml

```
<project name="test2" default="build" >
 <target name="build" >
  <fail message="Test NOK"/>
 </target>
</project>
```

5.2 Creating the test harness

The `harness.xml` script will just run the given test. Here is the content of this script:

```
<project name="harness" default="runTest">
 <target name="runTest">
  <!-- Just execute the Ant script -->
  <ant antfile="${testsuite.test.path}"/>
 </target>
</project>
```

5.3 Creating the MicroEJ Test-Suite launch

The `antTestsuite.xml` script will define the MicroEJ Test-Suite tasks, execute the testsuite and generate an HTML report.

The definition of the testsuite tasks is made by loading the antlib defined by the MicroEJ Test-Suite. The `testsuite` XML namespace is defined in the root tag of the Ant project.

```
<project name="antTestsuite" default="run"
 xmlns:testsuite="antlib:com.is2t.testsuite.ant">

 …

 <target name="testsuiteDefinition">
  <property name="testsuite.lib.dir" location="../testsuite-engine/lib"
 description="Path to the testsuite engine jars."/>

  <!-- Define testsuite tasks -->
  <taskdef uri="antlib:com.is2t.testsuite.ant" resource="com/is2t/testsuite/ant/
antlib.xml">
   <classpath>
    <fileset dir="${testsuite.lib.dir}" includes="*.jar"/>
   </classpath>
  </taskdef>
 </target>
</project>
```

- `The testsuite:testsuite task`: As explained on the section Using the test-suite Ant tasks, we will use the `testsuite:testsuite` task. Here is the call of the task:

  ```
  <testsuite:testsuite outputDir="results~" harnessScript="harness.xml">
     <testPath>
      <fileset dir="tests" includes="*.xml"/>
     </testPath>
    </testsuite:testsuite>
  ```

  Here is the JUnit report generated by the launch:

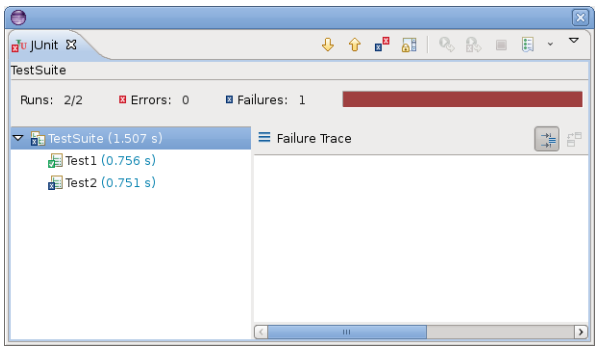
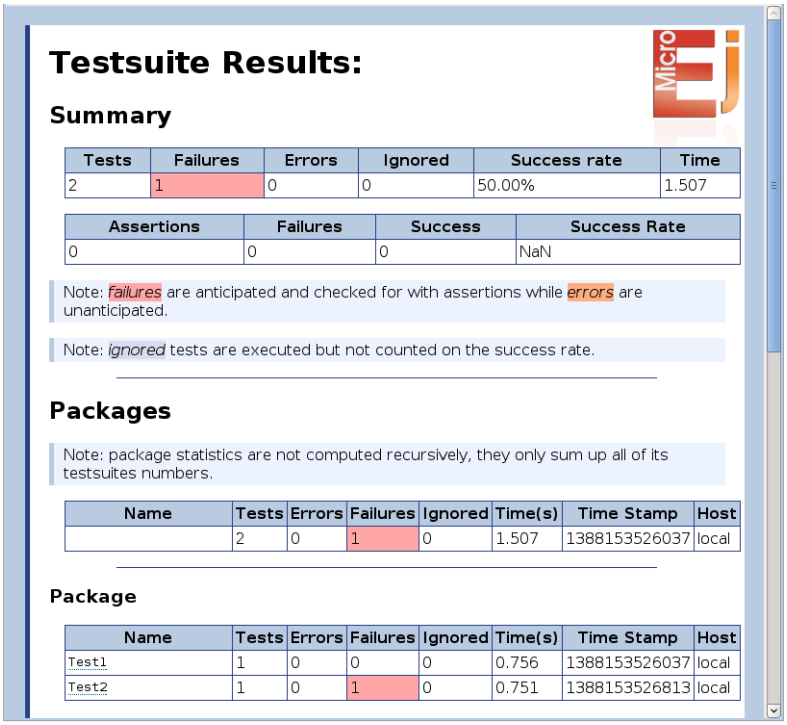


Figure 5.2. Example Ant - Final JUnit report

- `The testsuite:htmlReport task`: The `testsuite:htmlReport` task is used to generate the HTML report. A fileset containing the JUnit reports of each tests is given to the task.

  ```
  <testsuite:htmlReport>
     <fileset dir="${testsuite.report.dir}"> <!-- The 'testsuite.report.dir' property
   has been defined by the 'testsuite:testsuite' task -->
      <include name="**/*.xml"/> <!-- include unary reports -->
      <exclude name="*.xml"/> <!-- exclude global report -->
     </fileset>
     <report format="noframes" todir="${testsuite.report.dir}"/>
    </testsuite:htmlReport>
  ```

  Here is the HTML report (noframes) generated by the launch:



Testsuite Results:

Summary

Tests	Failures	Errors	Ignored	Success rate	Time
2	1	0	0	50.00%	1.507

Assertions	Failures	Success	Success Rate
0	0	0	NaN

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Note: *ignored* tests are executed but not counted on the success rate.

Packages

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

Name	Tests	Errors	Failures	Ignored	Time(s)	Time Stamp	Host
	2	0	1	0	1.507	1388153526037	local

Package

Name	Tests	Errors	Failures	Ignored	Time(s)	Time Stamp	Host
Test1	1	0	0	0	0.756	1388153526037	local
Test2	1	0	1	0	0.751	1388153526813	local

Figure 5.3. Example Ant - Final HTML report

6 Example - Deploying a Java test-suite

The goal of this section is to describe the steps to deploy a test suite environment on a simple application. For this example, we will start a simple MicroEJ application Test Suite on a IS2T Platform.

6.1 The MicroEJ environment

This example require to have a MicroEJ environment installed, and at least a platform on your MicroEJ repository.

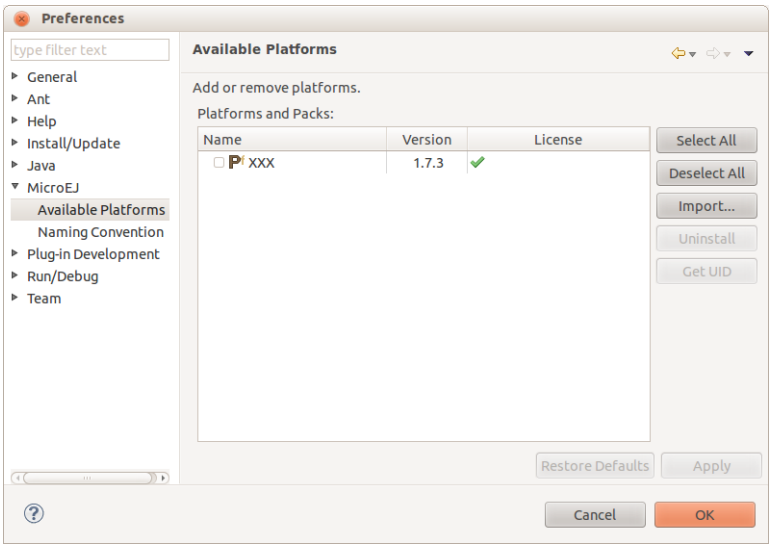


Figure 6.1. Example Java - Available platforms

6.2 Preview of the test-suite

For this example, we have four Java tests.

- `Test1.java`, which will fail.
- `Test2.java`, which will pass.
- `Test3.java`, which will do nothing.
- `Test4.java`, which will do infinitely.
- `Test5.java`, which will fail but is ignored.

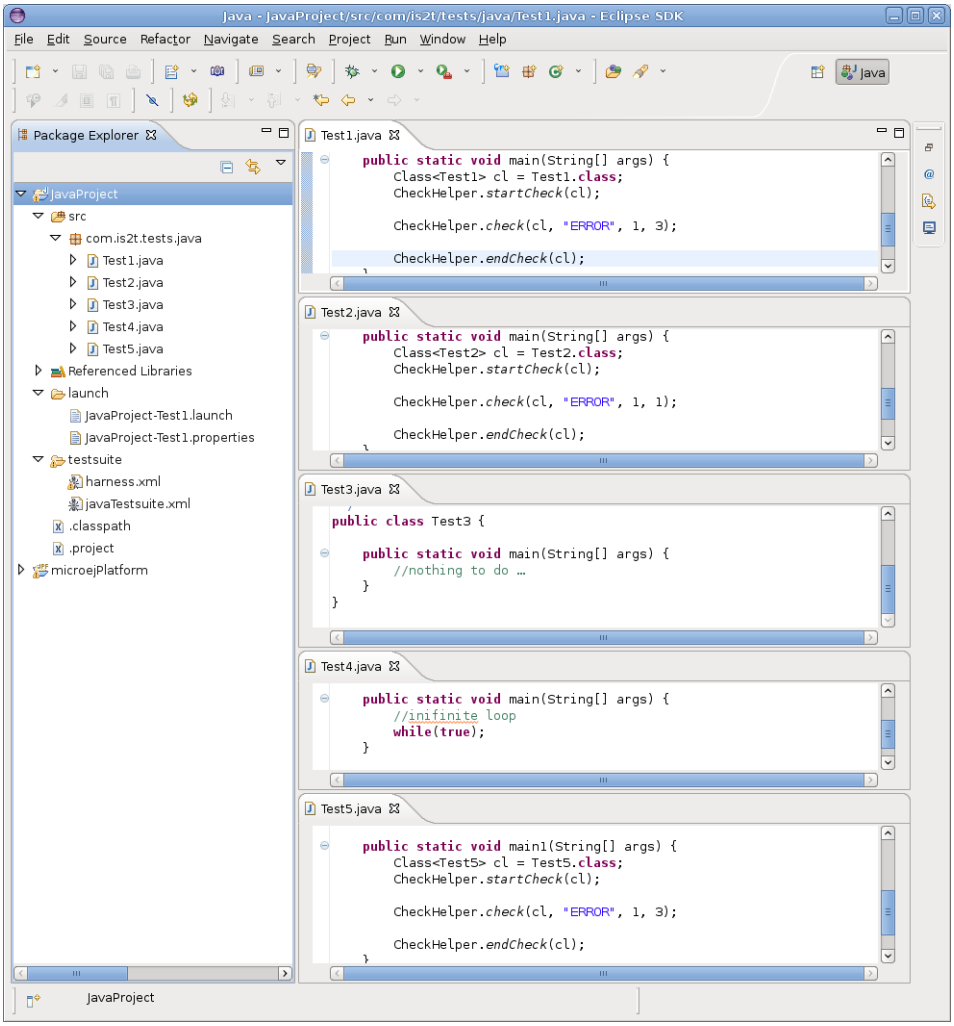


Figure 6.2. Example Java - The tree files

6.3 Creating the MicroEJ application launch

First, create a MicroEJ application launch, to retrieve all the common properties which are platform-dependants and common to every tests that will run through the test-suite. To do so, open the Eclipse menu **Run → Run Configuration...** and double click on MicroEJ Application.

Configure your launch as the following screenshots, for one of the tests, for example Test1.java. Do not forgot to check in the `Common Tab` to save the launch as a `Shared file` to be able to retrieve a property file from this launch.

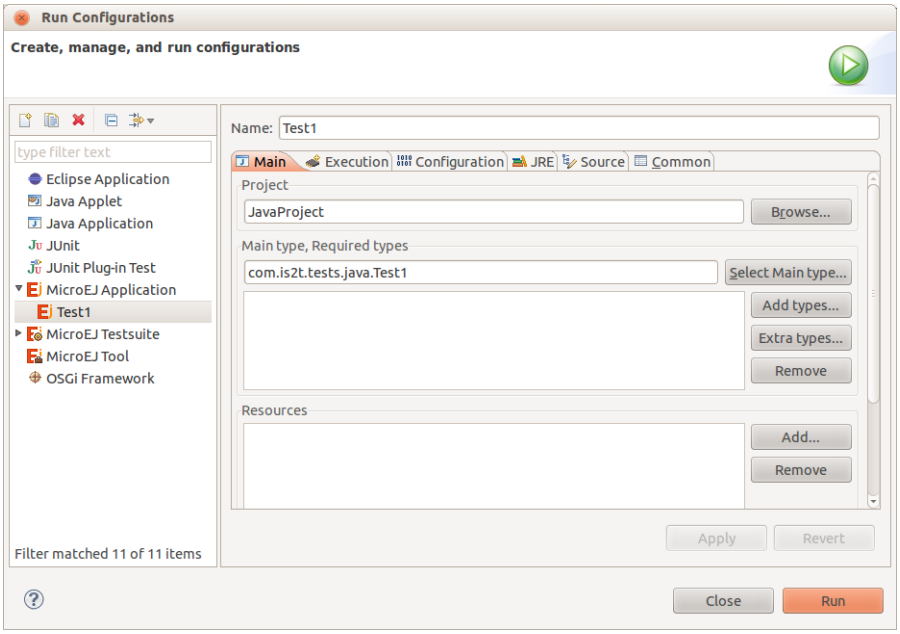


Figure 6.3. Example Java - Main Tab

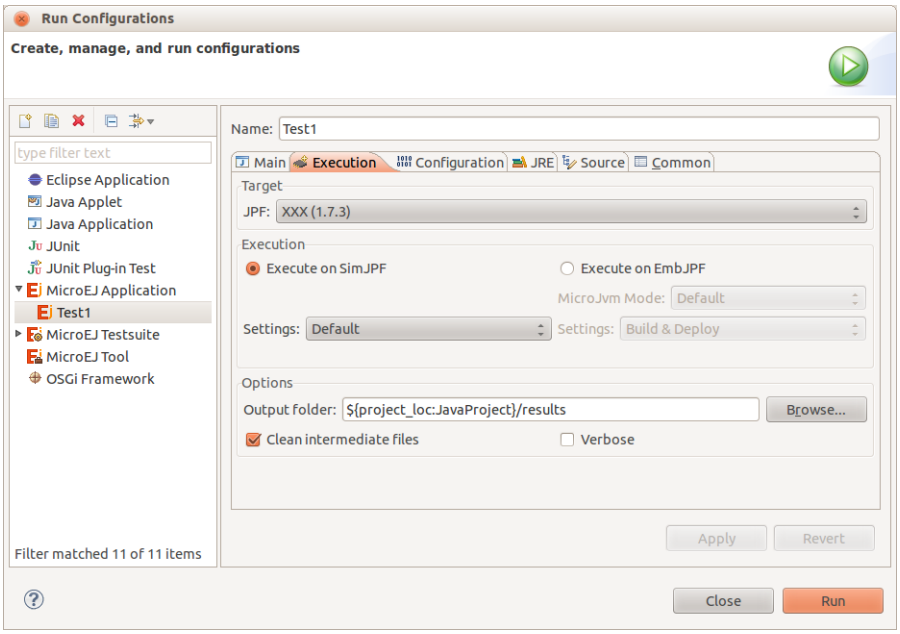


Figure 6.4. Example Java - Execution Tab

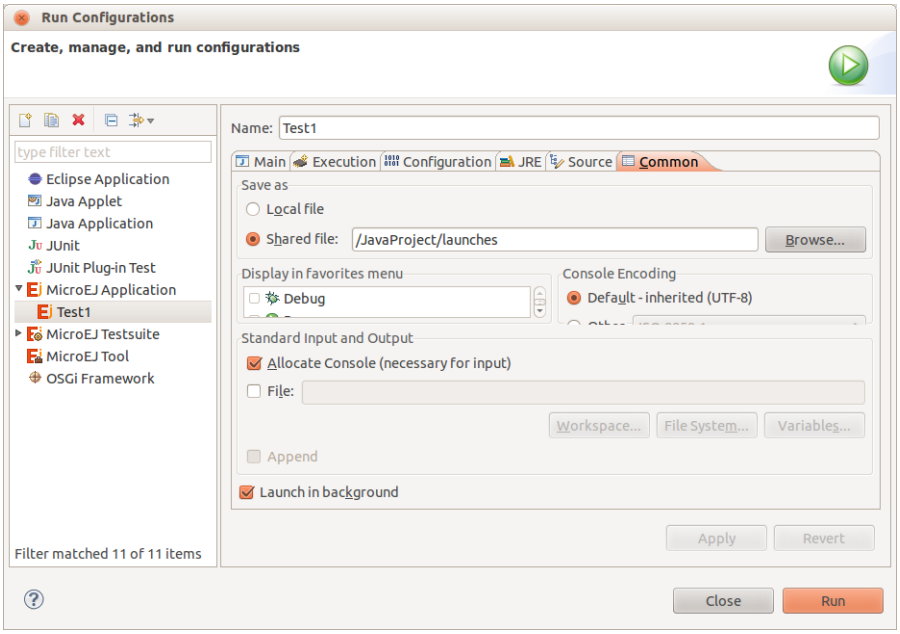


Figure 6.5. Example Java - Common Tab

Looking that we set up the test to be ran on simJPF, the script called by the launch will be `s3Default.microejLaunch`. When starting this launch, the expected output in the console are the following line. Looking we started the launch on Test1.java, we can see that the word `"FAILED"` as been wrote in the output.

```
====================[ INITIALIZATION STAGE      ]====================
====================[ LAUNCHING S3              ]====================
FAILED

LAUNCH SUCCESSFUL
```

6.4 The MicroEJ Test-Suite harness

The harness script is the script that will be started for every test. For this example, we will use the `File Trace Analyzer` provided by the MicroEJ Platform. The `File Trace Analyzer` is a tool allowing the user to parse a file searching for defined tags. Per default, it will search the following tags: `PASSED` and `FAILED`.

The harness consists in the following steps:

- Try to load the customs properties of the current test, located in the file saved in the property `testsuite.test.properties` sent by the test-suite engine. It try to loads them before loading the common properties, to override them if any conflicts.
- Try to load the common properties of all the tests, located in the file saved in the property `testsuite.common.properties` sent by the test-suite engine.
- Create a temporary file to store the standard output.
- Start the MicroEJ launch script `s3Default.microejLaunch` provided by the MicroEJ platform.
- Analyze the trace outputed in the temporary file to fail if the success tag is not encountered.

```
 <property file="${testsuite.test.properties}"/>
 <property file="${testsuite.common.properties}"/>

 <fail unless="platform.dir" message="Please set the 'platform.dir' property."/>

 <!-- Import the script that defines the trace analyzer tasks  -->
 <import file="${platform.dir}/scripts/traceAnalyzerDefinition.xml"/>

 <target name="runTest" depends="traceAnalyzer/definition">

  <!-- Create and clean microej.io.tmpdir (done by the workbench but not done when
 running from testsuite engine) -->
  <delete failonerror="false" dir="${microej.io.tmpdir}"/>
  <mkdir dir="${microej.io.tmpdir}"/>

  <!-- Prepare the file to save the trace -->
  <tempfile property="trace.file" prefix="trace" suffix=".txt"
 destdir="${microej.io.tmpdir}" deleteonexit="true"/>
  <record name="${trace.file}" action="start" />

  <!-- Run underlying the launch -->
  <ant antfile="${platform.dir}/scripts/s3Default.microejLaunch">
   <property name="application.classpath"
 value="${testsuite.test.classpath}${path.separator}${application.classpath}"/>
   <property name="application.main.class"  value="${testsuite.test.class}"/>
   <property name="output.dir" value="${testsuite.report.dir}/bin"/>
   <property name="basedir" value="${platform.dir}/scripts"/>
  </ant>

  <!-- Trace has ended -->
  <record name="${trace.file}" action="stop" />

  <!-- Analyze trace. Trace is fully generated: stop when EOF is reached -->
  <traceAnalyzer:fileTraceAnalyzer stopEOFReached="true" traceFile="${trace.file}"/>

  <!-- If the script reaches this point, test result is success -->
 </target>
```

6.5 Creating the MicroEJ Test-Suite launch

The `javaTestsuite.xml` script will define the MicroEJ Test-Suite tasks, execute the testsuite and generate an HTML report.

A path called `testsuite.tests.paths` is defined with the tests to execute and another path called `testsuite.ignored.tests.paths` is defined with the tests to ignore.

```
 <path id="testsuite.tests.paths">
 <fileset dir="${ant.dir.runTestsuite}/../bin/">
  <include name="**/T*.class"/>
 </fileset>
</path>

<path id="testsuite.ignored.tests.paths">
 <fileset dir="${ant.dir.runTestsuite}/../bin/">
  <include name="**/T*5.class"/>
 </fileset>
</path>
```

- The `testsuite:javaTestsuite task`: The testsuite:javaTestsuite task is defined by calling the `testsuite/definition` target defined in a script provided by the MicroEJ Platform. This task takes the MicroEJ Launch properties file as argument.

```
<!-- Import the script that defines the testsuite tasks  -->
<import file="${platform.dir}/scripts/testsuiteDefinition.xml"/>

<target name="run" depends="testsuite/definition">

 <!-- Launch test suite -->
 <testsuite:javaTestsuite
   outputDir="${ant.dir.runTestsuite}/../results~"
   harnessScript="${ant.dir.runTestsuite}/harness.xml"
   commonProperties="${microej.launch.propertyfile}"
 >
  <testPath refid="testsuite.tests.paths"/>
  <testIgnoredPath refid="testsuite.ignored.tests.paths"/>
 </testsuite:javaTestsuite>

 …
</target>
```

Here is the JUnit report generated by the launch:

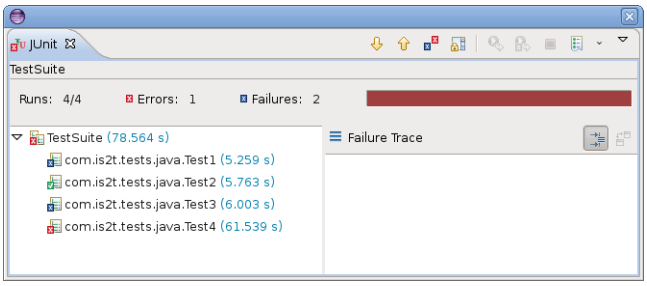
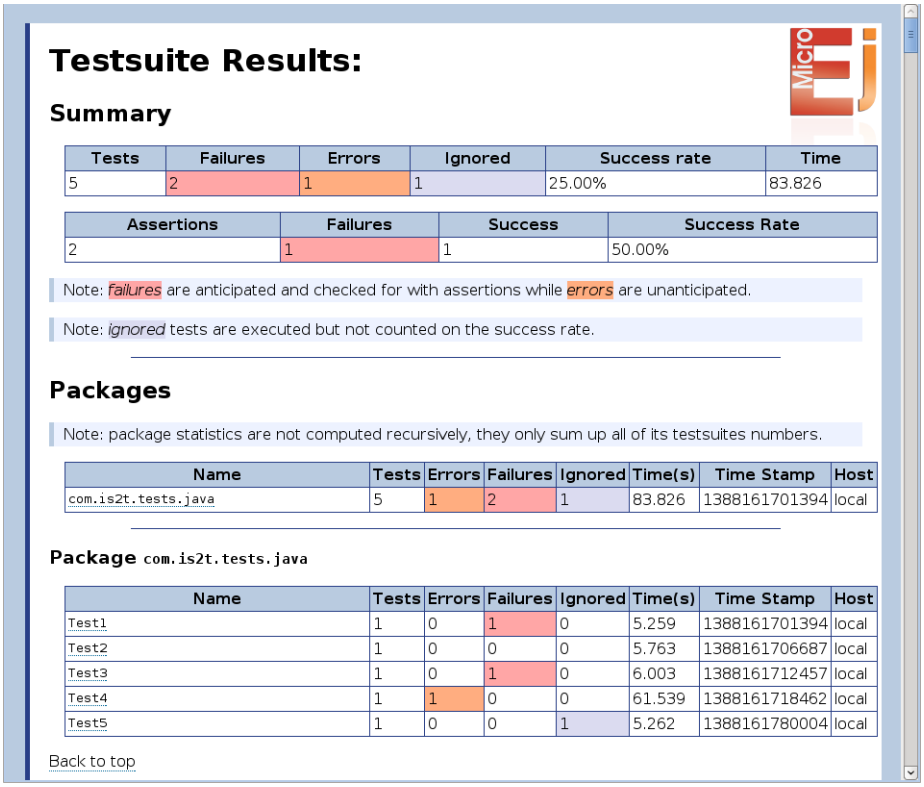


Figure 6.6. Example Java - Final JUnit report

- `The testsuite:htmlReport task`: The `testsuite:htmlReport` task is used to generate the HTML report. A fileset containing the JUnit reports of each tests is given to the task.

```
<!-- Generate HTML report -->
  <testsuite:htmlReport>
   <fileset dir="${testsuite.report.dir}">
    <include name="**/*.xml"/> <!-- include unary reports -->
    <exclude name="*.xml"/> <!-- exclude global report -->
   </fileset>
   <report format="noframes" todir="${testsuite.report.dir}"/>
  </testsuite:htmlReport>
```

Here is the HTML report (noframes) generated by the launch:



Testsuite Results:

Summary

Tests	Failures	Errors	Ignored	Success rate	Time
5	2	1	1	25.00%	83.826

Assertions	Failures	Success	Success Rate
2	1	1	50.00%

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Note: *ignored* tests are executed but not counted on the success rate.

Packages

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

Name	Tests	Errors	Failures	Ignored	Time(s)	Time Stamp	Host
com.is2t.tests.java	5	1	2	1	83.826	1388161701394	local

Package com.is2t.tests.java

Name	Tests	Errors	Failures	Ignored	Time(s)	Time Stamp	Host
Test1	1	0	1	0	5.259	1388161701394	local
Test2	1	0	0	0	5.763	1388161706687	local
Test3	1	0	1	0	6.003	1388161712457	local
Test4	1	1	0	0	61.539	1388161718462	local
Test5	1	0	0	1	5.262	1388161780004	local

Back to top

Figure 6.7. Example Java - Final HTML report

7 Using the trace analyzer

This section will shortly explains how to use the `Trace Analyzer`. The MicroEJ Test-Suite comes with an archive containing the `Trace Analyzer` which can be used to analyze the output trace of an application. It can be used from different forms;

- The `FileTraceAnalyzer` will analyze a file and research for the given tags, failing if the success tag is not found.
- The `SerialTraceAnalyzer` will analyze the data from a serial connection.

7.1 The TraceAnalyzer tasks options

Here is the common options to all TraceAnalyzer tasks:

- `successTag`: the regular expression with is synonym of success when found (by default `.*PASSED.*`).
- `failureTag`: the regular expression with is synonym of failure when found (by default `.*FAILED.*`).
- `verboseLevel`: int value between 0 and 9 to define the verbose level.
- `waitingTimeAfterSuccess`: waiting time (in s) after success before closing the stream (by default 5).
- `noActivityTimeout`: timeout (in s) with no activity on the stream before closing the stream. Set it to 0 to disable timeout (default value is 0).
- `stopEOFReached`: boolean value. Set to `true` to stop analyzing when input stream EOF is reached. If `false`, continue until timeout is reached (by default `false`).
- `onlyPrintableCharacters`: boolean value. Set to `true` to only dump ASCII printable characters (by default `false`).

7.2 The FileTraceAnalyzer task options

Here is the specific options of the FileTraceAnalyzer task:

- `traceFile`: path to the file to analyze.

7.3 The SerialTraceAnalyzer task options

Here is the specific options of the SerialTraceAnalyzer task:

- `port`: the comm port to open.
- `baudrate`: serial baudrate (by default 9600).
- `databits`: databits (5|6|7|8) (by default 8).
- `stopBits`: stopbits (0|1|3 for (1_5)) (by default 1).
- `parity`: `none` | `odd` | `event` (by default `none`).

8 Appendix

The goal of this section is to explain some tips and tricks that might be useful in your usage of the test-suite engine.

8.1 Specific custom properties

Some custom properties are specifics and retrieved from the test-suite engine in the custom properties file of a test.

- The `testsuite.test.name` property is the output name of the current test. Here are the steps to compute the output name of a test:
  - If the custom properties are enabled and a property named `testsuite.test.name` is find on the corresponding file, then the output name of the current test will be set to it.
  - Otherwise, if the running MicroEJ Test-Suite is a Java testsuite, the output name is set to the class name of the test.
  - Otherwise, from the path containing all the tests, a common prefix will be retrieved. The output name will be set to the relative path of the current test from this common prefix. If the common prefix cut the name of the test, then the output name will be set to the name of the test.
  - Finally, if multiples tests have the same output name, then the current name will be followed by _XXX, an underscore and an integer.
- The `testsuite.test.timeout` property allow the user to redefine the time out for each test. If it is negative or not an integer, then global timeout defined for the MicroEJ Test-Suite is used.