UnitCheck User's Manual

Michal Kebrt

Prague, April 2009

Contents

Chapter 1

User's manual

1.1 Command-line tool

Executable programs can be found in the `unitcheck/bin` directory – `unitcheck` for Unix environments and TODO for Windows). The following options are recognized, the bold ones are required.

`-t <classname>`	JUnit test class to be checked, fully-qualified class name must be used.
`-p <path1>:<path2>:...`	Colon-separated list of classpath elements that will be used by JPF VM for checking a test class.
`-s <path1>:<path2>:...`	Colon-separated list of sourcepath elements that will be used when reporting execution history. If not specified the history might be empty.
`-m <methodname>`	Name of a single method to be checked. By default all methods from a class are checked.
`-c <filename>`	Path to a JPF configuration file. All default JPF configuration parameters will be overriden with values from this file.
`-e`	Do not print the execution history when an error occurs (by default the history is printed).
`-a`	Trace also the standard Java libraries when an error occurs (by default the Java libraries are not traced).
`-d`	Print bytecode intructions in the execution history (by default bytecode is not printed).
`-j`	Do not filter JUnit and UnitCheck code out of the execution history (by default the filtering is on).
`-b <dirname>`	The base directory of UnitCheck. You will hardly ever need this option, it is set by wrapper scripts.
`-w`	Do not stop checking when an error occurs. It is not recommended to use this option.

The typical command for running UnitCheck follows.

```
$ ./unitcheck -p /mydir/myproject/classes -s /mydir/myproject/src
     -t cz.foo.Test1
```

1.2 Eclipse plugin

1.2.1 Installation

The plugin requires Eclipse IDE 3.4 or higher, it was successfully tested with versions 3.4.1 and 3.5M6. The plugin cannot be used with the version 3.3 or lower because it requires API that is not available in these old versions.

Before you can start using the plugin, it is necessary to properly install it to your Eclipse IDE. In order to do so, you have to access so called *update site*. This can be done in two ways:

- either you use the UnitCheck plugin update site at `http://aiya.ms.mff.c-uni.cz/unitchecking/plugin`,
- or you use the local update site archive (which you might e.g. build from the plugin sources).

In either case, follow these steps[1]:

1. Open the **Software Updates and Add-ons** dialog by selecting **Help|Software Updates** from the main menu.
2. Choose the **Available Software** tab.
3. Click **Add Site...** to define the update site.

   If you want to use the UnitCheck plugin update site, enter the URL of the site in the **Location** box. Click **OK** to confirm.

   In case you have the local update site archive (file named `unitcheck-eclipse-plugin-r` where $\star$ stands for the revision number), choose **Archive...** and in the displayed dialog select the file. Click **OK** to confirm.
4. Eclipse connects to the update site and offers you the features provided by the site. Check the **UnitCheck Eclipse Plugin** feature and click **Install...**.
5. In order to use the plugin, you have to agree with its license and continue by selecting **Finish**.
6. After the plugin is installed, acknowledge the Eclipse restart.

The plugin can be uninstalled using the same **Software Updates and Add-ons** dialog (select **Help|Software Updates** from the main menu). In the first **Installed Software** tab select **UnitCheck Eclipse Plugin** item and click the **Uninstall...** button. After you confirm that you agree with uninstallation, the plugin will be removed from your Eclipse IDE.

The plugin consists of two views that can be shown by opening the **Window|Show View|Other...** dialog and selecting the views in the **UnitCheck** category. You can place the views on your favourite locations in the Eclipse IDE. Both views will be described in next sections.

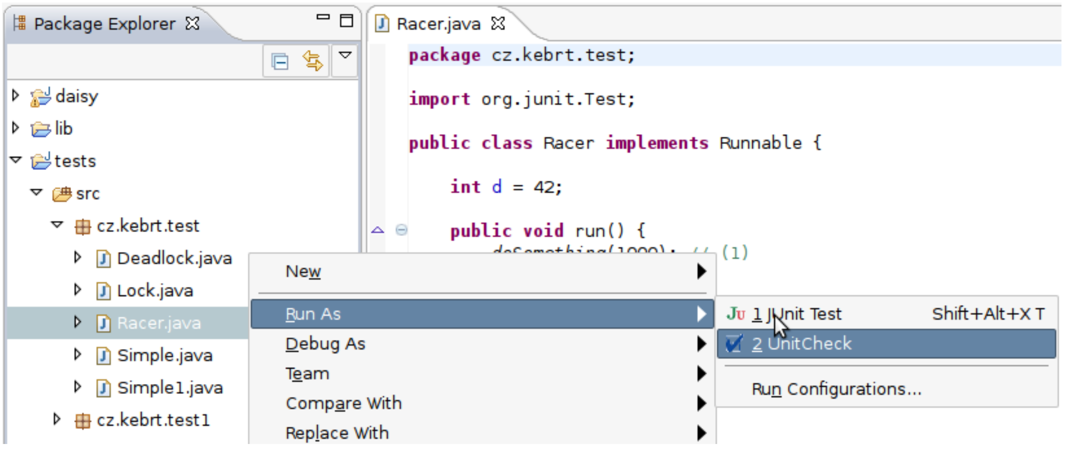
[1] Applies only to Eclipse 3.4, the plugin management is a little bit different in Eclipse 3.5.

1.2.2 Running

There are a couple of ways how to run UnitCheck on a selected JUnit test or a set of tests.

- One can right click a single test, directory or a whole project in the **Package Explorer** and select **Run As**|**UnitCheck** as Figure 1.1 shows. The **UnitCheck** run option is displayed only for items that contain at least one JUnit test. All test classes within selected element will then be checked.

Figure 1.1 Running UnitCheck from Package Explorer



```
package cz.kebrt.test;

import org.junit.Test;

public class Racer implements Runnable {

    int d = 42;

    public void run() {
        doSomething(1000); // (1)
```

- If a test class is opened in the Java editor, one may right click the source code and select **Run As**|**UnitCheck**. Only this single test class will be checked.

- The last way of running uses configurations that can be reached by selecting **Run** |**Run Configurations...** from the main menu. In the dialog right the **UnitCheck** category and select **New** which creates a new run configuration. The **Main** tab defines tests to be checked as Figure 1.2 illustrates.

  – Either a single test class can be checked. First a Java project must be selected using the **Browse...** button and after clicking the **Search** button a list of all classes from the project containing at least one JUnit test will be displayed. Only the selected class will be then checked.

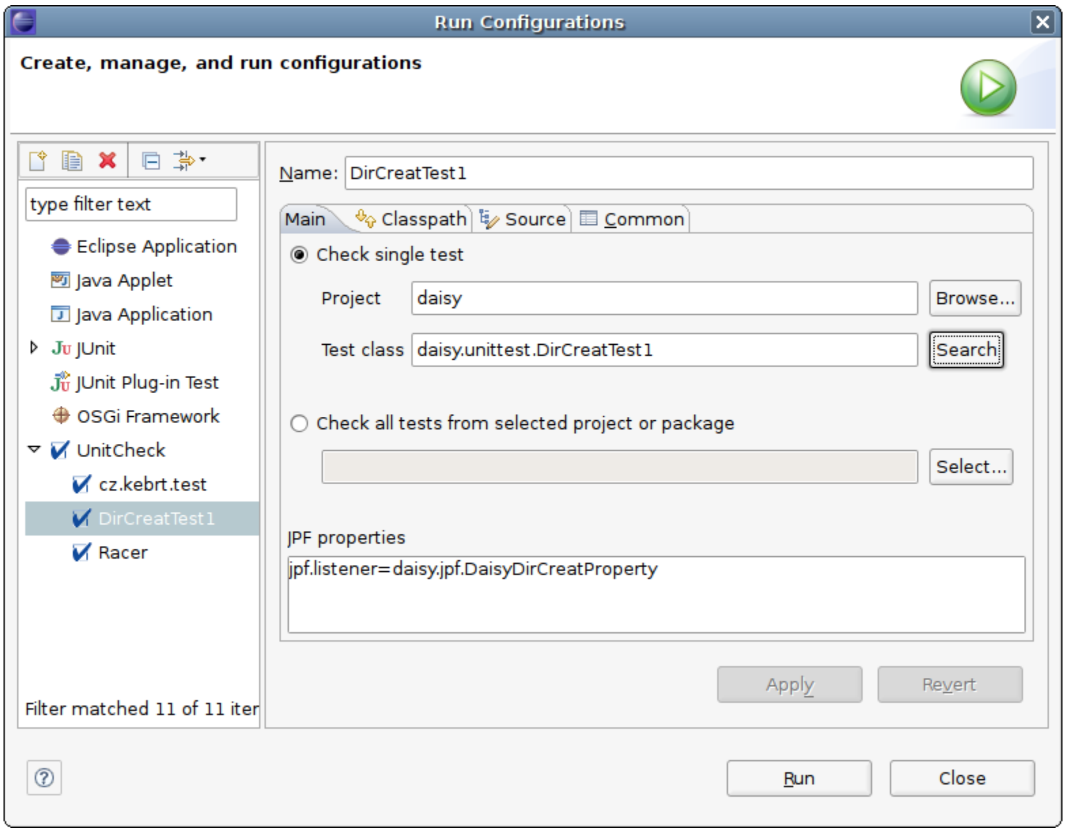
  – Or all tests from selected project or package can be checked. After clicking the **Select...** button the tree of all projects and packages containing at least one JUnit test will be displayed.

  JPF properties overriding default JPF properties can be specified in the textbox. Each one is written in a single row in the form of <*prop_name*>=<*prop_value*>. An example of a propery is `jpf.listener` for defining custom JPF listeners used when checking.

  You will hardly ever need to change anything in the standard Eclipse run tabs – **Classpath**, **Source** and **Common**. Classpath and sourcepath are based on the particular Java project's settings and are passed to JPF VM.

  If the configuration is ok the **Run** button starts checking of selected test classes.

Figure 1.2 Running UnitCheck from Run Configurations



Name: DirCreatTest1

(●) Check single test

Project: daisy

Test class: daisy.unittest.DirCreatTest1

( ) Check all tests from selected project or package

JPF properties

```
jpf.listener=daisy.jpf.DaisyDirCreatProperty
```

1.2.3 Result of checking

After the checking is started the **Console** view will contain the same output as the equivalent launch of the command-line UnitCheck program.

The **Check Summary** view incrementally displays classes that were checked in the current run. The view is split into two parts as Figure 1.3 shows. The first one contains the tree of all classes that were checked. Each class comprises of a list of test methods with icons indicating whether the checking of the particular method finished successfully or not. Double clicking any item in the tree opens its source code in the Java editor.

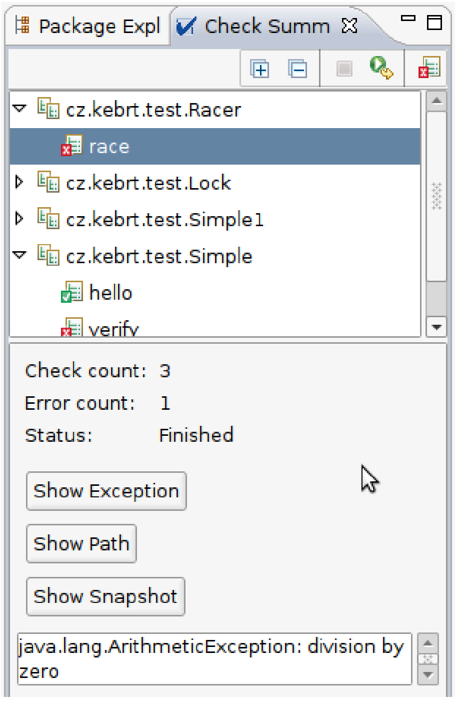
The second part of the view contains more information about checking results of a selected test method. In case an error was detected by UnitChecker three buttons for showing detailed information are displayed. The first button called **Show Exception** is enabled only for errors that were caused by an uncaught exception, it is disabled for violations of JPF properties. After clicking the button the exception stracktrace is printed in the **Path** view which will be described in the next section in more details. The **Show Path** and **Show Snapshot** buttons allow to read the complete execution history leading to the error and the state of all threads at the moment the error occured. A short of summary of the error can be found in the textbox below the buttons.

The right upper corner of the view contains a couple of buttons for manipulating with the tree of test classes. The buttons allow to

- expand and collapse all tree items,
- stop checking,

- rerun the previous checking,
- show only test methods that finished with an error.

Figure 1.3 Check Summary view



Check count: 3
Error count: 1
Status: Finished

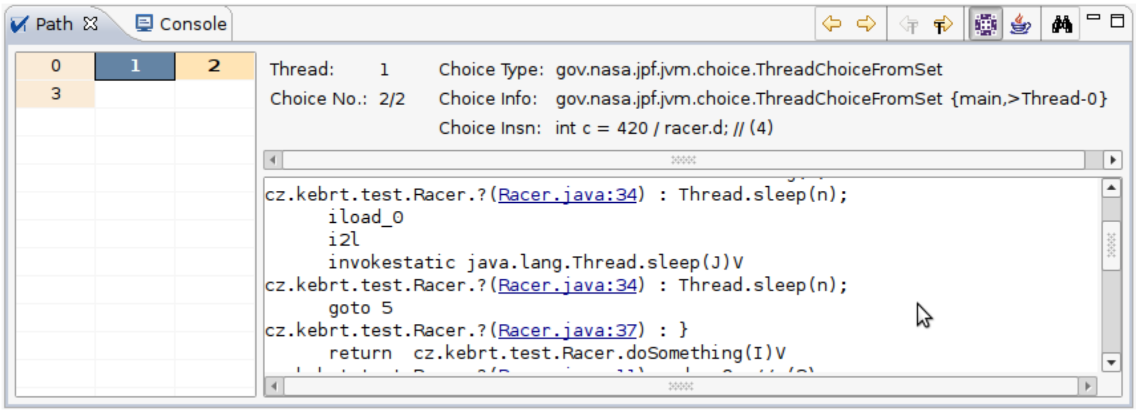
java.lang.ArithmeticException: division by zero

1.2.4 Inspecting error traces

This section applies only for situations where a test finished with an error. In such case the **Path** view can be used for detecting reasons that led to the error. In general the view displays the execution history provided by JPF but in more convenient way than the command-line program. The execution history, or path in other words, is split into a set of transitions which are listed in the left part of the view in a grid. The right part shows detailed information about the selected transition. Each transition is a result of so called *choice* in JPF, the choice usually produces more transitions as JPF backtracks to the *choice point* again and again. Therefore the user can see the following data about each transition.

- Number of the thread that executed the transition.
- Type of the choice (e.g. selection from a set of threads to be executed).
- Additional information about the choice (e.g. which thread was selected).
- Instruction that caused a choice to be created (i.e. the choice point).
- Number of the choice in the choice point (in the form of *choice_number/choice_total*).

Figure 1.4 Path view



```
Thread:     1     Choice Type:  gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet
Choice No.: 2/2   Choice Info:  gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {main,>Thread-0}
                  Choice Insn:  int c = 420 / racer.d; // (4)
```

```
cz.kebrt.test.Racer.?(Racer.java:34) : Thread.sleep(n);
        iload_0
        i2l
        invokestatic java.lang.Thread.sleep(J)V
cz.kebrt.test.Racer.?(Racer.java:34) : Thread.sleep(n);
        goto 5
cz.kebrt.test.Racer.?(Racer.java:37) : }
        return  cz.kebrt.test.Racer.doSomething(I)V
```

Transitions in the grid are coloured differently, those executed by the same thread are coloured with the same background colour. When a transition is selected all other transitions that were executed by the same thread are highlighted in bold. User can use either the keyboard arrows to move in the grid or arrow buttons in top upper corner of the view. The arrows marked with *T* jump only between transitions executed by the same thread.

Three other view buttons allow to

- show bytecode instructions along with the source code listing,
- trace standard Java libraries (can be combined with the previous button),
- search the code listing.

1.3 Ant task

For better integration of UnitCheck into existing projects the Ant task is also provided. This task should always be preferred to command-line program because it offers more convenient way of running JUnit under JPF.

After extracting `unitcheck-ant-task-rev*.zip` the following structure will appear:

- `lib` contains taks's binary code as well its dependencies in the form of JAR files,
- `src` and `classes` can be used for looking at example JUnit tests,
- `examples` contain Ant build files that use the UnitCheck task for checking sample tests.

Before the task can be used it must be properly defined in a build file as Example 1.1 shows. The task depends on a couple of JAR files located in its `lib` directory. According to Ant guidelines this way of defining task's dependencies is more flexible than using Ant's global `lib` directory.

Example 1.1 UnitCheck task definition & usage

```xml
<?xml version="1.0" ?>
<project name="myproject" default="mytarget" basedir=".">
  <property name="unitcheck.task.dir"
    location="<unitcheck-ant-task-dir>/lib" />

  <path id="unitcheckcp">
    <fileset dir="${unitcheck.task.dir}" includes="*.jar" />
  </path>

  <taskdef name="unitcheck"
    classname="cz.kebrt.unitcheck.ant.UnitCheckTask"
    classpathref="unitcheckcp"
  />

  <target name="mytarget">
    <property name="proj.dir" location="/project/daisy" />
    <unitcheck basedir="${unitcheck.task.dir}">
      <vmclasspath>
        <pathelement path="${proj.dir}/classes" />
      </vmclasspath>
      <vmsourcepath path="${proj.dir}/src,${proj.dir}/testsrc" />

      <jpfproperty key="jpf.listener"
        value="daisy.jpf.DaisyLockOrderProperty" />
      <jpfproperty key="search.properties"
        value="gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty" />

      <test class="daisy.unittest.CreatFileTest"
        method="testCreatFile" />
      <test class="daisy.unittest.CreatFileTest"
        method="testCreatLongFilename" />

      <batchtest>
         <fileset dir="${proj.dir}">
           <include name="**/*Test*.java" />
         </fileset>
      </batchtest>
    </unitcheck>
  </target>

</project>
```

The `unitcheck` task has the following attributes and subelements, the bold ones are required.

Attribute name	Default	Description
`basedir`		UnitCheck task's `lib` directory. The same directory was used in the task definition.

Attribute name	Default	Description
vmclasspathref		Reference to a classpath that will be used by JPF VM when checking. The vmclasspath element can be used instead.
vmsourcepathref		Reference to a sourcepath that will be used by JPF VM when reporting errors. If not specified the execution history might be empty. The vmsourcepath element can be used instead.
conffile		Path to a JPF configuration file. All default JPF configuration parameters will be overriden with values from this file. The jpfproperty elements can be used instead.
printpath	true	Print the execution history (path) when an error occurs.
printbytecode	false	Print the bytecode intructions along with the execution history.
filterjunittrace	true	Filter the JUnit and UnitCheck code out of the execution history.
tracejre	false	Trace the standard Java libraries in the execution history.

Element name	Description
vmclasspath	Classpath that will be used by JPF VM when checking. All standard Ant constructs for creatings path-like structures can be used within the element. The vmclasspathref attribute can be used instead.
vmsourcepath	Sourcepath that will be used by JPF VM when reporting errors. If not specified the execution history might be empty. All standard Ant constructs for creatings path-like structures can be used within the element. The vmsourcepathref attribute can be used instead.
jpfproperty	Allows to override default JPF configuration parameters. Each property has the key and value attributes.
test	Defines a JUnit test to be checked[2]. Each test must have a test class assigned. The method attribute allows to specify only a single test method, otherwise all test methods within the class will be checked.
batchtest	More tests to be checked can be specified using this element. All standard Ant constructs for defining filesets can be used within the element. Only the .java and .class files will be taken into account.

[2] Remember that the classpath containing the test and its dependencies must be configured using vmclasspath or vmclasspathref.