The Major Mutation System

Version 1.0.4

March 2013

Contents

1 Overview

Mutation analysis is widely applicable (e.g., for assessing testing and debugging techniques) and as a consequence also very differently used. Therefore, MAJOR divides the traditional mutation analysis process into two individual, consecutive parts:

1. Generate and embed mutants during compilation
2. Run the actual analysis, e.g., assess test-suite quality

For the first step, MAJOR provides a lightweight mutator, which is integrated in the *openjdk* Java compiler and configurable with its own domain specific language. For the second step, MAJOR also provides a default analysis back-end that extends Apache Ant's JUnit task.

2 The Mutation Compiler (Major-Javac)

Aiming at a lightweight and easy to use mutator, MAJOR extends the *openjdk* Java compiler. Within MAJOR's compiler, the conditional mutation approach [1] is implemented as an optional transformation of the abstract syntac tree (AST). In order to generate mutants, this transformation has to be enabled by setting the compiler option `-XMutator` — if the conditional mutation transformation is not enabled, then the compiler works exactly as if it were unmodified. The compile-time configuration of conditional mutation and the necessary runtime driver are externalized to avoid dependencies and to provide a non-invasive tool. As a consequence, MAJOR's compiler can be used as a compiler replacement in any Java-based development environment.

2.1 Conditional Mutation

Conditional mutation [1] generates mutants by transforming the abstract syntax tree (AST). It uses conditional expressions and statements to encapsulate all of the mutants and the original version of the program in the same basic block. Listings 1 gives an example, where the binary expression `a * x` is mutated. A concrete mutant can be triggered by setting the mutant identifier `M_NO` to the mutant's number at runtime. Note that the mutant identifier is externalized for two reasons: First, the externalization provides a unique access to the identifier. Second, the externalization ensures a consistent mutation numbering in multi class software systems.

```
1  public int eval(int x, int a, int b){
2      int y;
3
4      y = (M_NO==1)? a + x:
5          (M_NO==2)? a / x:
6          (M_NO==3)? a % x:
7                     a * x; // original
8      y += b;
9      return y;
10 }
```

Listing 1: Example method with an expression mutated with conditional mutation.

A mutant that is not covered (reached and executed) cannot be detected. Therefore, conditional mutation supports gathering additional information about the coverage of the mutations at runtime. Listing 2 shows the extension of the mutated binary expression, which provides the mutation coverage information with a call of the `COVERED` method. Note that this method is again externalized for consistency reasons.

```
1  public int eval(int x, int a, int b){
2      int y;
3
4      y = (M_NO==1)? a + x:
5          (M_NO==2)? a / x:
6          (M_NO==3)? a % x:
7          (M_NO==0 && COVERED(1,3))?
8              a * x : a * x; // original
9
10     y += b;
11     return y;
12 }
```

Listing 2: Gathering coverage information with conditional mutation.

2.2 Configuration

In order to avoid potential conflicts with future releases of the enhanced Java compiler, MAJOR extends the non-standardized `-X` options of the compiler. In order to use the mutation capabilities of MAJOR's compiler, the conditional mutation transformation has to be generally enabled at compile-time using the compiler option `-XMutator`. If the mutation step is enabled, MAJOR's compiler prints the number of generated mutants at the end of the compilation process and produces the log file *mutants.log*, which contains detailed information about each generated and embedded mutant.

To configure the process of generating mutants, MAJOR's compiler supports (1) compiler sub-options and (2) mutation scripts (use `javac -X` to see a description of all configuration options):

(1) `javac -XMutator:<sub-options>`

(2) `javac -XMutator=<mml filename>`

2.2.1 Compiler Options

MAJOR's compiler provides wildcards and a list of valid sub-options, which correspond to the names of the available mutation operators. For instance, the following two commands enable (1) all operators by means of the wildcard ALL and (2) only a subset of the available operators, namely AOR, ROR, and ORU:

(1) `javac -XMutator:ALL ...`

(2) `javac -XMutator:AOR,ROR,ORU ...`

Table 1 summarizes the mutation operators that are provided by MAJOR's compiler.

Table 1: Implemented mutation opertors.

	Description	Example
AOR	Arithmetic operator replacement	`a + b` ⟼ `a - b`
LOR	Logical Operator Replacement	`a ^ b` ⟼ `a | b`
COR	Conditional Operator Replacement	`a || b` ⟼ `a && b`
ROR	Relational Operator Replacement	`a == b` ⟼ `a >= b`
SOR	Shift Operator Replacement	`a >> b` ⟼ `a << b`
ORU	Operator Replacement Unary	`-a` ⟼ `~a`
STD	Statement Deletion Operator:	
	Delete (omit) a single statement	`foo(a,b)` ⟼ `<no-op>`
LVR	Literal Value Replacement:	
	Replace by a positive value,	`0` ⟼ `1`
	a negative value, and zero	`0` ⟼ `-1`

2.2.2 Mutation Scripts

Instead of using compiler options, MAJOR's compiler can interpret mutation scripts written in its domain specific language MML. These MML scripts enable a detailed definition and a flexible application of mutation operators. For example, the replacement list for every operator in an operator group can be specified and mutations can be enabled or disabled for certain packages, classes, or methods. Within the following example, the mutation process is controlled by the definitions of the compiled script file *myscript.mml.bin*:

- `javac -XMutator="pathToFile/myscript.mml.bin"`

Note that MAJOR's compiler interprets pre-compiled script files. Use the script compiler mmlc to syntactically and semantically check, and compile a MML script file. MAJOR's domain specific language MML is described in detail in the subsequent Section 3.

2.3 Driver Class

The conditional mutation components of MAJOR's compiler reference an external driver to gain access to the mutant identifier (`M_NO`). Additionally, the driver has to furnish the mutation coverage method (`COVERED`). Listing 3 shows an example of a simple driver class

that provides both the mutant identifier and the mutation coverage method. The identifier and the coverage method must be implemented in a static context to avoid any overhead caused by polymorphism and instantiation.

```
1  package major.mutation;
2
3  import java.util.*;
4
5  public class Config{
6      public static int M_NO=0;
7      public static Set<Integer> covSet = new TreeSet<Integer>();
8
9      // Record coverage information
10     public static boolean COVERED(int from, int to){
11         for(int i=from; i<=to; ++i){
12             covSet.add(i);
13         }
14         return false;
15     }
16     // Reset the coverage information
17     public static void reset(){
18         covSet.clear();
19     }
20     // Get (copied) list of all covered mutants
21     public static List<Integer> getCoverageList(){
22         return new ArrayList<Integer>(covSet);
23     }
24 }
```

Listing 3: Driver class providing the mutant identifier `M_NO` and coverage method `COVERED`.

In order to keep MAJOR non-invasive, the driver class does **not** have to be available on the classpath during compilation. MAJOR does not try to resolve the driver class at compile-time but instead assumes that the mutant identifier and the coverage method will be available in this class at runtime. Thus, the mutants can be generated without having a driver class available during compilation.

2.4 Log File of Generated Mutants

MAJOR's compiler generates the log file *mutants.log*, which provides detailed information about the generated mutants. This log file uses the csv format and the colon (`:`) as separator — it contains one row per generated mutant, where each row in turn contains 7 columns with the following information:

1. Mutants' unique number (id)
2. Name of the applied mutation operator
3. Original operator symbol
4. Replacement operator symbol

5. Fully qualified name of the mutated method
6. Line number in original source file
7. Visualization of the applied transformation (`from |==> to`)

The following example gives the log entry for a ROR mutation that has the mutant id 11 and is generated for the method `classify` (line number 18) of the class `Triangle`:

```
11:ROR:<=(int,int):<(int,int):Triangle@classify:18:a <= 0 |==> a < 0
```

2.5 Integration into Apache Ant's build.xml

MAJOR's compiler can be used standalone, but also in build systems, such as Apache Ant. Considering, for example, the following `compile` target in a build.xml file:

```
<target name="compile" depends="init" description="Compile">
    <javac srcdir="src"
          destdir="bin"
            debug="yes">
    </javac>
</target>
```

In order to use MAJOR's compiler without any changes to an existing Java environment, add the following 3 options to the `compile` target:

```
<property name="mutOp" value=":NONE"/>
<target name="compile" depends="init" description="Compile">
    <javac srcdir="src"
          destdir="bin"
            debug="yes"

             fork="yes"
       executable="pathToMajor/javac">
    <compilerarg value="-XMutator${mutOp}"/>

    </javac>
</target>
```

There is no need to duplicate the entire target since MAJOR's compiler can also be used for regular compilation. The following three commands illustrate how the `compile` target shown above can be used to: (1) compile without mutation, (2) compile with mutation using compiler options, and (3) compile with mutation using a MML script:

(1) `ant compile`

(2) `ant -DmutOp=":ALL" compile`

(3) `ant -DmutOp="=pathToFile/myscript.mml.bin" compile`

Note that the `mutOp` property provides a default value (`:NONE`) iff this property is not set on the command line.

3 The Major Mutation Language (Major-Mml)

To enable both fundamental research as well as empirical studies, MAJOR is designed to support a wide variety of configurations by means of its own domain specific language, called MML. Generally, a MML script contains a sequence of an arbitrary number of statements, where a statement represents one of the following entities:

- Variable definition
- Invocation of a mutation operator
- Replacement definition
- Definition of an own operator group
- Line comment

While the first three statements are terminated by a semicolon, an operator definition is encapsulated by curly braces and a line comment is terminated by the end-of-line.

3.1 Statement Scopes

In order to support the mutation of a certain package, class, or method within a software system, MML provides statement scopes for replacement definitions and operator invocations. Figure 1 depicts the definition of a statement scope, which can cover software units at different levels of a given hierarchy, from a specific method up to an entire software package. Note that a statement scope is optional as indicated by the first rule of Figure 1. If no statement scope is provided, the corresponding replacement definition or operator call is applied to the highest level of the given hierarchy. The scope's corresponding software entity, that is package, class, or method, is determined by means of its fully qualified name, which is referred to as flatname. Such a flatname can be either provided within delimiters (DELIM) or by means of a variable identified by IDENT.

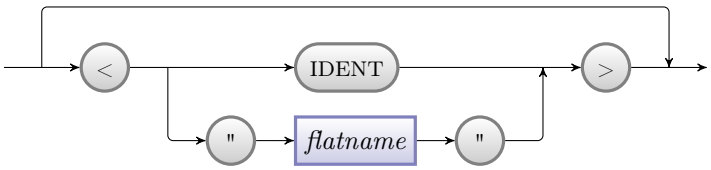

Figure 1: Syntax diagram for the definition of a statement scope.

The grammar rules for assembling a flatname are shown in Figure 2. The naming conventions for valid identifiers (IDENT) are based on those of the Java programming language due to the fact that a flatname identifies a certain entity within a Java software system. The following three examples are, for instance, valid flatnames for a package, class, and method:

- `"java.lang"`
- `"java.lang.System"`
- `"java.lang.System@exit"`

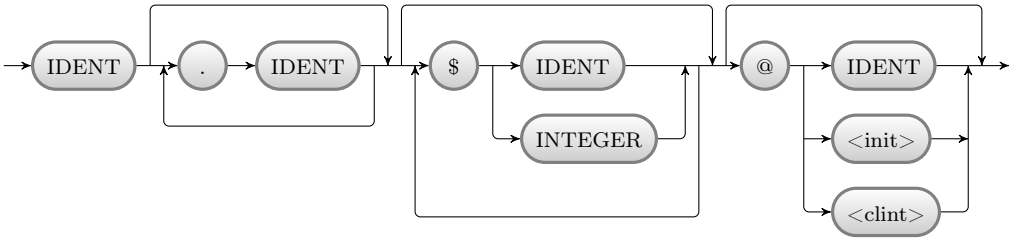

Figure 2: Syntax diagram for the definition of a flatname.

Note that the syntax definition of a flatname also supports the identification of innerclasses and constructors, consistent with the naming conventions of the Java compiler. For Example, the subsequent definitions address an inner class, a constructor, and a static class initializer:

- `"foo.Bar$InnerClass"`
- `"foo.Bar@<init>"`
- `"foo.Bar@<clinit>"`

3.2 Overriding and Extending Definitions

In principle, mutation operators can be enabled (`+`), which is the default if the flag is omitted, or disabled (`-`) and this behavior can be defined for each scope. In the following example, the AOR mutation operator is generally enabled for the package `org` but, at the same time, disabled for the class `Foo` within this package:

```
+AOR<"org">
-AOR<"org.Foo">
```

Note that the flag for enabling or disabling operators is optional — the default flag (`+`) for enabling operators improves readability but can be omitted.

With regard to replacement definitions, there are two different possibilities: Individual replacements can be added (`+`) to an existing list or the entire replacement list can be overridden (`!`), where the latter possibility represents the default case if this optional flag is omitted. The following example illustrates this feature, where the general definition of replacements for the package `org` is extended for the class `Foo` but overriden for the class `Bar`. Note that the replacement lists that are actually applied to the package and classes are given in comments.

```
 BIN(*)<"org">     -> {+,/}    // (*) -> {+,/}
+BIN(*)<"org.Foo"> -> {%}      // (*) -> {+,/,%}
!BIN(*)<"org.Bar"> -> {-}      // (*) -> {-}
```

3.3 Operator Groups

To prevent further code duplication due to repeated executions of equal definitions for several scopes (i.e., the same replacements or enabled mutation operators for several packages, classes, or methods), the MML provides the possibility to declare own operator groups. Such a group may in turn contain any statement that is valid in the context of the MML, except for a call of another operator group. An operator group is defined by means of a unique identifier and its statements are enclosed by curly braces, as shown in the following example:

```
myGroup {
    ...
}
```

3.4 Script Examples

Listing 4 shows a simple example of a mutation script that includes the following tasks:

- Define specific replacement lists for AOR and ROR
- Invoke the AOR and ROR operators on reduced lists
- Invoke the LVR operator without restrictions

```
1  // Define own replacement list for AOR
2  BIN(*) -> {/,%};
3  BIN(/) -> {*,%};
4  BIN(%) -> {*,/};
5
6  // Define own replacement list for ROR
7  BIN(>)  -> {<=,!=,==};
8  BIN(==) -> {<,!=,>};
9
10 // Enable and invoke mutation operators
11 AOR;
12 ROR;
13 LVR;
```

Listing 4: Simple script to define replacements for the AOR and ROR mutation operators and to enable AOR, ROR, and LVR on the root node.

The more enhanced script in Listing 5 exploits the scoping capabilities of the MML in line 8 and 13-20, and takes, additionally, advantage of the possibility to define a variable in line 11 to avoid code duplication in the subsequent scope declarations. Both features are useful if only a certain package, class, or method shall be mutated in a hierarchical software system. Finally, the example in Listing 6 visualizes the grouping feature, which is useful if the same group of operations (replacement definitions or mutation operator invocations) shall be applied to several packages, classes, or methods.

```
// Definitions for the root node
BIN(>=)->{TRUE,>,==};
BIN(<=)->{TRUE,<,==};
BIN(!=)->{TRUE,<,> };
LVR;

// Definition for the package org
ROR<"org">;

// Variable definition for the class Foo
foo="org.x.y.z.Foo";

// Scoping for replacement lists
BIN(&&)<foo>->{LHS,RHS,==,FALSE};
BIN(||)<foo>->{LHS,RHS,!=,TRUE };

// Scoping for mutation operators
-LVR<foo>;
ROR<foo>;
COR<foo>;
```

Listing 5: Enhanced mutation script with scoping and variable definition.

```
myOp{
    // Definitions for the operator group
    BIN(>=)->{TRUE,>,==};
    BIN(<=)->{TRUE,<,==};
    BIN(!=)->{TRUE,<,> };
    BIN(&&)->{LHS,RHS,==,FALSE};
    BIN(||)->{LHS,RHS,!=,TRUE };
    // Mutation operators enabled in this group
    ROR;
    COR;
}

// Calls of the defined operator group
myOp<"org">;
myOp<"de">;
myOp<"com">;
```

Listing 6: Mutation script with a definition of an own mutation operator group and corresponding calls for different scopes.

4 The Mutation Analysis Back-end (Major-Ant)

MAJOR provides a default back-end for mutation analysis, which extends the Apache Ant `junit` task. Therefore, this back-end can be used to evaluate existing JUnit tests. Note that MAJOR does not support forking a JVM when executing JUnit tests, meaning that the `fork` option must not be set to `true` — forking will be supported in a future release.

4.1 Setting up a Mutation Analysis Target

Most software projects that are build with Apache Ant provide a `test` target, which executes the corresponding unit tests. Even if no such target exists, it can be easily set up to execute a set of given unit tests. The following code snippet shows an exemplary `test` target (See http://ant.apache.org/manual/Tasks/junit.html for a detailed description of the options used in the `junit` task):

```
<target name="test" description="Run all unit test cases">
    <junit printsummary="true"
             showoutput="true"
          haltonfailure="true">

        <formatter type="plain" usefile="true"/>
        <classpath path="bin"/>
        <batchtest fork="no">
            <fileset dir="test">
                <include name="**/*Test*.java"/>
            </fileset>
        </batchtest>
    </junit>
</target>
```

In order to enable mutation analysis in MAJOR's enhanced version of the `junit` task, the option `mutationAnalysis` has to be set to `true`. For the sake of clarity, it is advisable to duplicate an existing `test` target, instead of parameterizing it, and to create a new target, e.g., called `mutation-test` (See Section 4.3 for recommended configurations):

```
<target name="mutation-test" description="Run mutation analysis">
    <junit printsummary="false"
             showoutput="false"
          haltonfailure="true"

       mutationAnalysis="true"
              resltFile="results.csv">

        <classpath path="bin"/>
        <batchtest fork="no">
            <fileset dir="test">
                <include name="**/*Test*.java"/>
            </fileset>
        </batchtest>
    </junit>
</target>
```

4.2 Configuration Options for Mutation Analysis

MAJOR enhances the `junit` task with additional options to control the mutation analysis process. The available, additional, options are summarized in Table 2.

Table 2: Additional configuration options for Major-Ant's JUnit task

	Description	Values	Default
mutationAnalysis	Enable mutation analysis	`[true|false]`	`false`
coverage	Enable mutation coverage	`[true|false]`	`true`
timeoutFactor	Base timeout factor for test runtime	`<int>`	`8`
sort	Enable sort of test cases	`[original|random|` `sort_classes|` `sort_methods|` `sort_hybrid_classes|` `sort_hybrid_methods]`	`original`
threshold	Threshold in milliseconds for sort_hybrid_xx	`<int>`	`50`
excludeFile	Exclude mutants which ids are listed in this file (1 id per row)	`<String>`	`null`
resultFile	Export detailed runtime information to this file (csv)	`<String>`	`null`
killDetailsFile	Export kill details for each mutant to this file (csv)	`<String>`	`null`
exportCovMap	Export mutation coverage map	`[true|false]`	`false`
covMapFile	File name for mutation coverage map (csv)	`<String>`	`covMap.csv`
testMapFile	File name for mapping of test id to test name (csv)	`<String>`	`testMap.csv`

4.3 Performance Optimization

During the mutation analysis process, the provided JUnit tests are repeatedly executed. For performance reasons, consider the following advices when setting up the mutation analysis target for a JUnit test suite:

- Turn off logging output (options `showsummary`, `showoutput`, etc.)
- Do not use result formatters (nested task `formatter`, especially the `usefile` option)

For performance reasons, especially due to frequent class loading and thread executions, the following JVM options are recommended when using MAJOR's `junit` task:

- `-XX:ReservedCodeCacheSize=128M`
- `-XX:MaxPermSize=256M`

5 Step by Step Tutorial

This sections provides a step by step tutorial on how to use MAJOR for:

- Configure the mutant generation with MML scripts (Section 5.1)
- Generate mutants with MAJOR's compiler (Section 5.2)
- Run mutation analysis with MAJOR's back-end (Section 5.3)

5.1 Prepare and Compile a Mml Script

To enable an easy integration into existing build infrastructures and prevent from modifying existing experiment setups, MAJOR supports a detailed specification of the mutation process by means of it's domain specific language MML. Suppose only relational and conditional binary operators shall be mutated for the method `classify` of the class `triangle.Triangle`. The following MML script (*cor_ror.mml*) is suitable for this purpose:

```
1  targetOp{
2      // Use sufficient replacements for ROR
3      BIN(>)->{>=,!=,FALSE};
4      BIN(<)->{<=,!=,FALSE};
5      BIN(>=)->{>,==,TRUE};
6      BIN(<=)->{<,==,TRUE};
7      BIN(==)->{<=,>=,FALSE,LHS,RHS};
8      BIN(!=)->{<,>,TRUE,LHS,RHS};
9
10     // Use sufficient replacements for COR
11     BIN(&&)->{==,LHS,RHS,FALSE};
12     BIN(||)->{!=,LHS,RHS,TRUE};
13
14     // Enable ROR and COR operators
15     COR;
16     ROR;
17 }
18 // Define the target method
19 target="triangle.Triangle@classify";
20
21 // Call defined operator group for specified target
22 targetOp<target>;
```

Listing 7: MML script to generate COR and ROR mutants for the `classify` method.

Section 3 provides a detailed description of the syntax and capabilities of the domain specific language MML. The MAJOR framework also provides a MML compiler (mmlc) that checks a MML script and compiles it into a binary representation. Given the MML script (*cor_ror.mml*), the MML compiler is invoked with the following command:

```
tutorial$ mmlc cor_ror.mml cor_ror.mml.bin
tutorial$
```

Note that the second argument is optional — if omitted, the compiler will add *.bin* to the name of the provided script file, by default.

5.2 Generate Mutants

In order to generate mutants based on the MML script prepared in Section 5.1, the compiled script (*cor_ror.mml.bin*) has to be passed to MAJOR's compiler.

5.2.1 Compiling Standalone

```
tutorial$ javac -XMutator=cor_ror.mml.bin -d bin src/triangle/Triangle.java
#Generated Mutants: 79 (41 ms)
tutorial$
```

5.2.2 Compiling with Apache Ant

If the source files shall be compiled using Apache Ant, the `compile` target of the corresponding *build.xml* file needs to be adapted to use MAJOR's compiler and to provide the necessary compiler option (See Section 2.5 for further details):

```
<property name="mutOp" value=":NONE"/>
<target name="compile" depends="init" description="Compile">
    <javac srcdir="src" destdir="bin" debug="yes"
           fork="yes" executable="pathToMajor/javac">
        <compilerarg value="-XMutator${mutOp}"/>
    </javac>
</target>
```

Given the compiled MML script and the adapted *build.xml* file, use the following command to mutate and compile the source files:

```
tutorial$ ant -DmutOp="=cor_ror.mml.bin" compile

compile:
    [javac] Compiling 1 source file to tutorial/bin
    [javac] #Generated Mutants: 79 (26 ms)

BUILD SUCCESSFUL
Total time: 1 second
tutorial$
```

5.2.3 Inspecting Generated Mutants

If mutation has been enabled (i.e., the `-XMutator` option is used). MAJOR's compiler reports the number of generated mutants. Additionally, it produces the log file (*mutants.log*) that contains detailed information about the generated mutants (see Section 2.4 for a description of the format). The following example shows the log entries for the first 3 generated mutants:

```
tutorial$ head -3 mutants.log
1:ROR:<=(int,int):<(int,int):Triangle@classify:18:a <= 0 |==> a < 0
2:ROR:<=(int,int):==(int,int):Triangle@classify:18:a <= 0 |==> a == 0
3:ROR:<=(int,int):TRUE(int,int):Triangle@classify:18:a <= 0 |==> true
tutorial$
```

5.3 Analyze Mutants

Now, to use MAJOR's `junit` task to perform the mutation analysis of a given test suite, the *build.xml* file has to provide a corresponding target. The following `mutation-test` target enables the mutation analysis and exports the results to *results.csv* and *killed.csv* (See Section 4 for further details):

```
<target name="mutation-test" description="Run mutation analysis">
    <echo message="Running mutation analysis ..."/>
    <junit printsummary="false" showoutput="false" mutationAnalysis="true"
           resultFile="results.csv" killDetailsFile="killed.csv">
        <classpath path="bin"/>
        <batchtest fork="false">
            <fileset dir="test">
                <include name="**/*Test*.java"/>
            </fileset>
        </batchtest>
    </junit>
</target>
```

The following command invokes the `mutation-test` target:

```
tutorial$ ant mutation-test

mutation-test:
     [echo] Running mutation analysis ...
    [junit] MAJOR: Mutation analysis enabled
    [junit] MAJOR: ------------------------------------------------------------
    [junit] MAJOR: Run mutation analysis with 1 individual test
    [junit] MAJOR: ------------------------------------------------------------
    [junit] MAJOR: 1/1 - TestSuite (4ms / 79):
    [junit] MAJOR: 527 (70 / 79 / 79) -> AVG-RTPM: 6ms
    [junit] MAJOR: Mutants killed / live: 70 (70-0-0) / 9
    [junit] MAJOR: ------------------------------------------------------------
    [junit] MAJOR: Summary:
    [junit] MAJOR:
    [junit] MAJOR: Total runtime:  0.5 seconds
    [junit] MAJOR: Mutation score: 88.61%
    [junit] MAJOR: Mutants killed / live: 70 (70-0-0) / 9
    [junit] MAJOR: ------------------------------------------------------------
    [junit] MAJOR: Export runtime results (to results.csv)
    [junit] MAJOR: Export mutant kill details (to killed.csv)

BUILD SUCCESSFUL
Total time: 1 second
tutorial$
```

References

[1] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proceedings of the 6th ACM/IEEE International Workshop on Automation of Software Test*, AST '11, pages 50–56. ACM Press, 2011.

[2] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering*, ISSRE '12. IEEE Computer Society, 2012.

[3] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 612–615. IEEE Computer Society, 2011.