The Major Mutation Framework

Version 1.1.0

January 2014

Contents

1 Overview

Major is a complete mutation analysis framework that divides the mutation analysis process into two individual, consecutive steps:

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

For the first step, Major provides a lightweight mutator, which is integrated in the openjdk Java compiler. For the second step, Major provides a default analysis back-end that extends Apache Ant's JUnit task.

1.1 Installation

This section describes how to install Major for use from the command line or Ant. The installation is simple — all you need is included in the Major release package!

- Download the Major framework from http://mutation-testing.org/major.zip.
- Unzip `major.zip` to create the `major` directory.
- Optionally, update your environment and prepend Major's `bin` directory to your execution path (`PATH` variable).

The `major` directory provides the following content:

```
major
├── bin .......................... Executables for Major's components
│   ├── ant
│   ├── javac
│   └── mmlc
├── config ....................... Archive and sources of Major's driver
├── doc .......................... The manual of the current version
│   └── major.pdf
├── example ...................... Examples for using Major with Ant and standalone
│   ├── ant
│   │   ├── ...
│   │   └── run.sh
│   ├── standalone
│   │   ├── ...
│   │   └── run.sh
│   └── runAll.sh
├── lib .......................... All libraries used by Major
└── mml .......................... Example Mml files
```

1.2 How to get started

Verify that you are using Major's compiler by running `javac -version`. The output should be the following:

```
major$ javac -version
javac 1.7.0-Major-v1.1.0
```

Suppose you want to mutate a class `MyClass.java`. The following command mutates and compiles `MyClass.java` using all mutation operators:

```
major$ javac -XMutator:ALL MyClass.java
#Generated Mutants: 90 (28 ms)
```

Note that `javac` must refer to Major's compiler, which always prints the number of generated mutants when the `-XMutator` flag is enabled. Additionally, Major's compiler produces a log file (`mutants.log`) for all generated mutants.

All generated mutants are embedded in the compiled class files. The `example` directory provides two examples on how to use mutants to perform mutation analysis on test suites:

- `ant`: mutation analysis using Apache Ant.
- `standalone`: mutation analysis using Major's driver standalone.

Execute `runAll.sh` within the `example` directory to run all examples or `run.sh` in a subdirectory for a particular example. Section 2 provides a step by step tutorial on how to use Major for a project using Apache Ant, and the subsequent sections describe Major's components and configuration options in detail:

- Section 3 provides details about Major's compiler.
- Section 4 describes Major's DSL (Mml).
- Section 5 provides details about Major's default mutation analysis back-end.

2 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 2.1).
- Generate mutants with Major's compiler (Section 2.2).
- Run mutation analysis with Major's back-end (Section 2.3).

You can find all files and the triangle program, which are used in this tutorial, in the `example` and `mml` directories (see Section 1.1).

2.1 Prepare and compile a Mml script

Major supports a detailed specification of the mutation process by means of its 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 (`tutorial.mml`) is suitable for this purpose:

```
targetOp{
    // Use sufficient replacements for ROR
    BIN(>)->{>=,!=,FALSE};
    BIN(<)->{<=,!=,FALSE};
    BIN(>=)->{>,==,TRUE};
    BIN(<=)->{<,==,TRUE};
    BIN(==)->{<=,>=,FALSE,LHS,RHS};
    BIN(!=)->{<,>,TRUE,LHS,RHS};

    // Use sufficient replacements for COR
    BIN(&&)->{==,LHS,RHS,FALSE};
    BIN(||)->{!=,LHS,RHS,TRUE};

    // Enable ROR and COR operators
    COR;
    ROR;
}
// Define the target method
target="triangle.Triangle@classify";

// Call defined operator group for specified target
targetOp<target>;
```

Listing 2.1: Mml script to generate COR and ROR mutants for the `classify` method.

Section 4 provides a detailed description of the syntax and capabilities of the domain specific language Mml. The Major framework provides a compiler (`mmlc`) that compiles Mml scripts into a binary representation. Given the Mml script `tutorial.mml`, the `mmlc` compiler is invoked with the following command:

```
major$ mmlc tutorial.mml tutorial.mml.bin
```

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

2.2 Generate mutants

To generate mutants based on the compiled Mml script `tutorial.mml.bin` (see Section 2.1), the compiled script has to be passed as an argument to Major's compiler.

2.2.1 Compiling from the command line

Use the `-XMutator` option to mutate and compile from the command line:

```
major$ javac -XMutator=tutorial.mml.bin -d bin src/triangle/Triangle.java
#Generated Mutants: 79 (25 ms)
```

2.2.2 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 3.4 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 `tutorial.mml.bin` script and the adapted `build.xml` file, use the following command to mutate and compile the source files:

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

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

BUILD SUCCESSFUL
Total time: 1 second
```

2.2.3 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 3.2.1 for a description of the format). The following example shows the log entries for the first 3 generated mutants:

```
major$ head -3 mutants.log
1:ROR:<=(int,int):<(int,int):triangle.Triangle@classify:20:a <= 0 |==> a < 0
2:ROR:<=(int,int):==(int,int):triangle.Triangle@classify:20:a <= 0 |==> a == 0
3:ROR:<=(int,int):TRUE(int,int):triangle.Triangle@classify:20:a <= 0 |==> true
```

Major also supports the export of generated mutants to individual source files — see 3.2.2 for more details.

2.3 Analyze mutants

The `build.xml` file has to provide a suitable `mutation.test` target to use Major's mutation analysis back-end, which performs the mutation analysis for a given test suite. The following `mutation.test` target enables the mutation analysis and exports the results to `results.csv` and `killed.csv` (see Section 5 for further details):

```
<target name="mutationr.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>
```

Using MAJOR's version of `ant`, the following command invokes the `mutation.test` target:

```
major$ 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
```

As configured in the `build.xml` file, the results of the mutation analysis are exported to the files `results.csv` and `killed.csv`, which provide the following information:

- `results.csv`: Detailed runtime information and mutation analysis results for each executed test.
- `killed.csv`: The reason why a mutant was killed — i.e., assertion failure, exception, or timeout.

3 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.

3.1 Configuration

MAJOR extends the non-standardized `-X` options to avoid potential conflicts with future releases of the Java compiler. 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`. 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>`

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.

3.1.1 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 3.1 summarizes the mutation operators that are provided by MAJOR's compiler.

3.1.2 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

Table 3.1: 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, a negative value, and zero	0 ⟼ 1 0 ⟼ -1

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 4.

3.2 Logging and exporting generated mutants

3.2.1 Log file for generated mutants

MAJOR's compiler generates the log file `mutants.log`, which provides detailed information about the generated mutants and uses a colon (:) as separator. The log file 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
```

3.2.2 Export of generated mutants

Major also supports the export of each generated mutant to an individual source file — this feature is disabled by default. If enabled, Major duplicates the original source file for each mutant, injects the mutant in the copy, and exports the resulting faulty copy. Major reads the following two properties that control the export of generated mutants (default values are given in parentheses):

- `-J-Dmajor.export.mutants=[true|false] (false)`
- `-J-Dmajor.export.directory=<directory> (./mutants)`

Major automatically creates the export directory and parent directories if necessary.

Note that, if you are mutating a large code base, exporting all mutants to individual source files increases the compilation time and requires significantly more disk space than the log file.

3.3 Driver class

Major references an external driver at runtime to gain access to a mutant identifier (`M_NO`) and a method that monitors mutation coverage (`COVERED`). Listing 3.1 shows an example of a simple driver class that provides both the mutant identifier and the mutation coverage method. Note that the mutant identifier and the coverage method must be implemented in a static context to avoid any overhead caused by polymorphism and instantiation.

The archive and source files of the default driver implementation is provided in the `config` directory. Note that 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 provided by the driver class at runtime. Thus, Major's compiler is non-invasive and the mutants can be generated without having a driver class available during compilation.

```java
package major.mutation;

import java.util.*;

public class Config{
    public static int M_NO=0;
    public static Set<Integer> covSet = new TreeSet<Integer>();

    // Record coverage information
    public static boolean COVERED(int from, int to){
        for(int i=from; i<=to; ++i){
            covSet.add(i);
        }
        return false;
    }
    // Reset the coverage information
    public static void reset(){
        covSet.clear();
    }
    // Get (copied) list of all covered mutants
    public static List<Integer> getCoverageList(){
        return new ArrayList<Integer>(covSet);
    }
}
```

Listing 3.1: Driver class providing the mutant identifier `M_NO` and coverage method `COVERED`.

3.4 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:

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

To use MAJOR's compiler without any further changes to your environment, add the following 3 options to the `compile` target:

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

             fork="yes"
       executable="pathToMajor/bin/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`) if this property is not set on the command line.

4 The Major Mutation Language (Major-Mml)

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.

4.1 Statement scopes

MML provides statement scopes for replacement definitions and operator invocations to support the mutation of a certain package, class, or method within a program. Figure 4.1 depicts the definition of a statement scope, which can cover software units at different levels of granularity — from a specific method up to an entire package. Note that a statement scope is optional as indicated by the first rule of Figure 4.1. If no statement scope is provided, the corresponding replacement definition or operator call is applied to the root package. The scope's corresponding 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.

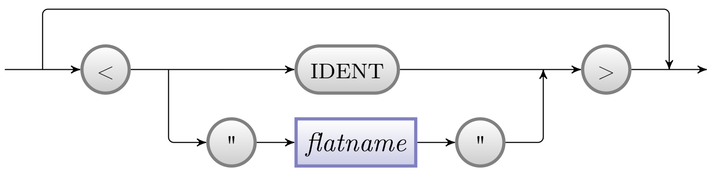

Figure 4.1: Syntax diagram for the definition of a statement scope.

Figure 4.2 shows the grammar rules for assembling a flatname. The naming conventions for valid identifiers (IDENT) are based on those of the Java programming language due to

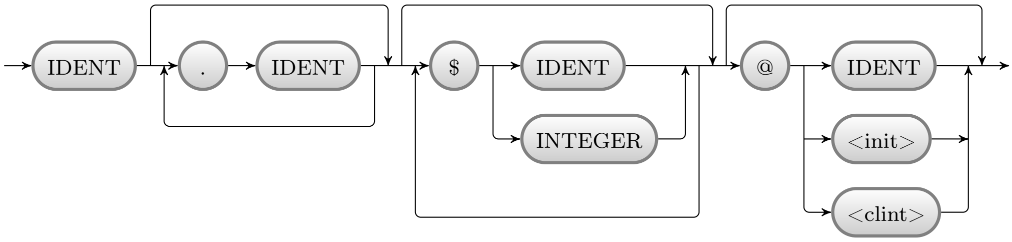

Figure 4.2: Syntax diagram for the definition of a flatname.

the fact that a flatname identifies a certain entity within a Java program. The following three examples are, for instance, valid flatnames for a package, class, and method:

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

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>"`

4.2 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 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`. The replacement lists that are effectively applied to the package and classes are given in comments.

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

4.3 Operator groups

To prevent code duplication due to the repetition of equal definitions for several scopes (i.e., the same replacements or enabled mutation operators for several packages, classes, or methods), 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 {
    BIN(*) -> {+,/};
    AOR;
}
```

4.4 Script examples

Listing 4.1 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

```
// Define own replacement list for AOR
BIN(*) -> {/,%};
BIN(/) -> {*,%};
BIN(%) -> {*,/};

// Define own replacement list for ROR
BIN(>)  -> {<=,!=,==};
BIN(==) -> {<,!=,>};

// Enable and invoke mutation operators
AOR;
ROR;
LVR;
```

Listing 4.1: 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 4.2 exploits the scoping capabilities of 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.

```
1 // Definitions for the root node
2 BIN(>=)->{TRUE,>,==};
3 BIN(<=)->{TRUE,<,==};
4 BIN(!=)->{TRUE,<,> };
5 LVR;
6
7 // Definition for the package org
8 ROR<"org">;
9
10 // Variable definition for the class Foo
11 foo="org.x.y.z.Foo";
12
13 // Scoping for replacement lists
14 BIN(&&)<foo>->{LHS,RHS,==,FALSE};
15 BIN(||)<foo>->{LHS,RHS,!=,TRUE };
16
17 // Scoping for mutation operators
18 -LVR<foo>;
19 ROR<foo>;
20 COR<foo>;
```

Listing 4.2: Enhanced mutation script with scoping and variable definition.

Finally, the example in Listing 4.3 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.

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

5 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 currently 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.

5.1 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>
```

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 5.3 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>
```

5.2 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 5.1.

5.3 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:

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

Table 5.1: 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`
exportKillMap*	Export mutation kill map	`[true|false]`	`false`
killMapFile	File name for mutation kill map	`<String>`	`killMap.csv`
testMapFile	File name for mapping of test id to test name (csv)	`<String>`	`testMap.csv`

*Note: this option leads to the execution of every test on every covered mutant!

Bibliography

[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. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *Proceedings of the 7th IEEE International Workshop on Mutation Analysis*, Mutation '12, pages 720–725. IEEE Computer Society, 2012.

[3] 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, pages 11–20. IEEE Computer Society, 2012.

[4] 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.