STAMP

Deliverable D3.5

Final report for online-test amplification

Project Number	:	731529
Project Title	:	STAMP
Deliverable Type	:	Report

Deliverable Number	:	D3.5
Title of Deliverable	:	Final report for online-test amplification
Dissemination Level	:	Public
Version	:	1.5
Latest version	:	https://github.com/STAMP-project/docs-forum/blob/master/docs/d35_final_report_for_online_test_amplification.pdf
Contractual Delivery Date	:	M36 - November 30, 2019
Contributing WPs	:	WP 3
Editor(s)	:	Xavier Devroey, TU Delft
Author(s)	:	Luca Andreatta, Engineering Mohamed Boussaa, ActiveEon Pouria Derakhshanfar, TU Delft Xavier Devroey, TU Delft Ciro Formisano, Engineering Daniele Gagliardi, Engineering Arie van Deursen, TU Delft Pasquale Vitale, Engineering Andy Zaidman, TU Delft
Reviewer(s)	:	Benoit Baudry, KTH Caroline Landry, INRIA Brice Morin, SINTEF

Abstract

This deliverable presents the tools developed within the work package 3 of the STAMP project, dedicated to online amplification. Those tools include the Botsing framework to process log data, extract common usages of objects, and generate crash reproducing test cases using a search-based approaches. And RAMP, an extension of EvoSuite for Runtime AMPlification, able to generate test cases using model seeding, a novel seeding approach used to generate more relevant objects during the search-based test case generation process.

Revision History

Version	Type of Change	Author(s)
1.0	Initial version	Xavier Devroey, TU Delft
1.1	Adding chapter on Botsing optimization and user guide of Maven plugins	Ciro Formisano, Engineering
1.2	Adding information on Botsing architecture and developer guide	Pouria Derakhshanfar, TU Delft
1.3	Adding information on RAMP empirical evaluation	Mohamed Boussaa, ActiveEon
1.4	Review changes and updating empirical evaluations	Xavier Devroey, TU Delft
1.5	Update after comments from the reviewers	Xavier Devroey, TU Delft

Contents

Glossary

This glossary presents the terminology used across the different deliverables of work package 3.

Botsing: meaning *crash* in Dutch, is the name of the framework for online test amplification developed in WP3. It includes a crash reproduction tool and many other extensions to for crash reproduction and search-based test case generation.

Code instrumentation: code instrumentation in Botsing consists in the injection of probes into the bytecode of a Java application to monitor and log the runtime behavior of specific classes.

RAMP: Runtime AMPlification is the extension of EvoSuite developed in WP3 that uses model seeding (i.e., commonly observed behavior of objects) for test generation.

JCrashPack: JCrashPack is a benchmark containing 200 crashes to assess crash replication tools capabilities [7].

Model seeding: in search-based software testing, seeding consist in providing external information to the search algorithm to help the exploration of the search space. Model seeding, developed within STAMP, is the seeding of transition systems (a formalism similar to state machines describing a dynamic behavior) to a search based test case generation algorithm to improve its capabilities. The models represent the common usages of classes and allow the generate objects with sequences of method calls, representing a common behavior in the application. In Botsing, models are learned from the source code (static analysis) and from the logs produced by the execution of the system (dynamic analysis using instrumentation) using n-gram inference.

Stack trace: a (crash) stack trace is a piece of log data usually denoting a crash failure. A stack trace provides information on the exception thrown and on the propagation of that exception trough the stack of method calls.

Stack trace preprocessing: stack traces can contain redundant and useless information preventing to automatically reproduce the associated crash. The preprocessing allows to filter stack traces to keep only relevant information.

Introduction

This deliverable presents the different tools developed within WP3, dedicated to online test amplification. Online test amplification automatically extracts information from logs collected in production in order to generate new tests that can replicate failures, crashes, anomalies and outlier events.

Log data analysis. During Task 3.1, we investigated the current state-of-the-art in the field of log analysis by doing a literature survey and identified the current state-of-the-practice of the STAMP partners regarding logging by performing interviews. We reported those findings in D3.1. During the literature survey, we found that the results of the various works do not lend themselves well to a comparison, and suggest future steps of research to overcome this lack of clarity. Furthermore, we suggest areas of improvement regarding the applicability and feasibility of analysis techniques.

From the interviews of the STAMP partners, we found out that there is a significant disparity in logging practices. Logging practices usually come from the experience of the developers and code reviewers and are rarely documented. Logs are mainly used and enhanced during the debugging of the source code after a problem occurred and has been reported. In particular, (i) there is no uniform way of logging that we found among the STAMP project partners, and even different systems of the partners might have different logging approaches; (ii) some of the STAMP partners for not have access to logging data due to privacy issues; (iii) some STAMP partners do not log details of exceptional situations in their default log.

Those findings and the knowledge that stack traces, denoting exceptional behavior of the software, are reported in the issue tracker motivated to focus our effort on stack traces for crash replication (Task 3.3). And static and dynamic analysis of the software to learn state machine models (Task 3.2) and generate test cases with common and uncommon behavior (Task 3.3 and 3.4).

Learning state machine models. During Task 3.2, we devised and implemented a model generation tool able to generate state machines (i.e., transition systems) representing the common behavior of the classes of the software under test. This tool is described in chapter 2 and generates models that are used in the subsequent Tasks 3.3 and 3.4 to enhance crash reproduction and test case generation. A report of the characteristics of the models generated during the empirical evaluations performed in Tasks 3.3 and 3.4 is available at the end of chapter 2.

Test Case Generation. During Task 3.3, we developed a tool for crash reproduction as part of the Botsing framework (see chapter 1 and chapter 4). Botsing crash reproduction exploits information available in the crash stack trace. We developed, evaluated, and refined (see D3.3 and D3.4) novel fitness functions to guide test case generation algorithms toward the generation of tests directly usable by developers to find the cause of the crash and fix the bugs. Those developments were driven by our findings after performing a large empirical evaluation of crash reproduction with EvoCrash, a legacy implementation of a search-based crash reproduction tool, redeveloped from scratch in Botsing in Task 3.3, and JCrashPack, the first benchmark for Java crash reproduction, developed during Task 3.2 and described in D3.2.

During Task 3.4, we used the models generated from static analysis and dynamic execution of the code to generate unit tests. Models are seeded to the search process to generate test cases with object usages close to what can be observed in the software under test. The tool developed within STAMP is an extension (fork) of EvoSuite: EvoSuite Runtime AMPlification (RAMP), described in chapter 5.

Runtime log enhancement. During Task 3.2, we also extended the code instrumentation mechanism of Botsing by injecting additional probes into the bytecode of a Java application to monitor and log the runtime behavior of specific classes (see D3.4). This injection allows to enhance the crash reproduction capabilities of Botsing by improving the testability of the code and providing more information to guide the search. The evaluation of the new fitness function using our new code instrumentation mechanism is available in chapter 1.

Also, this extension of the code instrumentation allowed us to generate class integration tests, thanks to the definition of a new coverage criterion and the fitness function able to guide a search-based test case generation process to satisfy this criterion. The extension, new coverage criterion, new fitness function, and the results of the empirical evaluation of class integration test case generation are available in D3.4.

Finally, after the large scale empirical evaluation reported in D3.2, we defined several actions that can be performed to preprocess a stack trace and enhance the crash reproduction capabilities of Botsing. Those preprocessing steps have been implemented and integrated with the Botsing framework and are described in chapter 3.

Botsing optimization. During Task 3.5, we identified speeding up strategies, both for the usage and runtime execution, from the uses of Botsing made by the STAMP industrial partners. We defined the requirements for a parallel version of Botsing, described in chapter 6, and coordinated its development.

The remainder of this document is structured as follows:

chapter 1 - The Botsing framework presents the Botsing framework developed within WP3, with its general architecture and general guidelines for contributors.

chapter 2 - Botsing model generation presents the module related to the generation of model representing common object usages, used to seed a search-based test case or crash reproduction process.

chapter 3 - Botsing preprocessing present the module related to the cleanup and preprocessing of the stack traces.

chapter 4 - Botsing crash reproduction presents the module related to crash reproduction using a given Java stack trace.

chapter 5 - RAMP presents a tool to generate unit tests using commonly observed object behavior.

chapter 6 - Botsing optimization presents a tool to run parallel Botsing crash reproduction process and identifies other possible optimizations using distributed genetic algorithms.

Relation to WP3 tasks

In Table 1 we summarize how the prototypes and results reported in this deliverable relate to the 5 tasks of WP3.

Table 1: WP3 tasks reported in this deliverable

Task 3.1	This task ended at M6.
Task 3.2	We measure and reported general statistics on the models generated using model seeding (chapter 2)
Task 3.3	We have evaluated the effect of our code instrumentation extension on the crash reproduction capabilities of Botsing (chapter 4).
Task 3.4	We have evaluated RAMP with model seeding in order to improve test case generation with commonly observed behavior (chapter 5).
Task 3.5	We have documented and described our parallel version of Botsing crash reproduction (chapter 6).

Summary of Artifacts

1. **Botsing framework** (see chapter 1)
    - Link: https://github.com/STAMP-project/botsing/
    - Contact: Pouria Derakhshanfar (TUD), Xavier Devroey (TUD), Luca Andreatta (Eng), Pasquale Vitale (Eng)
2. **Botsing parallel execution** (see chapter 6)
    - Link: https://github.com/STAMP-project/botsing-parallel
    - Contact: Pasquale Vitale (Eng)
3. **RAMP** (see chapter 5)
    - Link: https://github.com/STAMP-project/evosuite-ramp
    - Contact: Pouria Derakhshanfar (TUD), Xavier Devroey (TUD)
4. **Botsing crash reproduction tutorial** (see WP4 T4.4)
    - Link: https://github.com/STAMP-project/botsing-demo
    - Contact: Pouria Derakhshanfar (TUD), Xavier Devroey (TUD)
5. **RAMP tutorial** (see WP4 T4.4)
    - Link: https://github.com/STAMP-project/evosuite-ramp-tutorial
    - Contact: Pouria Derakhshanfar (TUD), Xavier Devroey (TUD)

Chapter 1

The Botsing framework

Botsing is a framework for Java crash reproduction and test generation using runtime information of the software under test, developed in the context of the STAMP project. It completely re-implements and extends EvoCrash, a crash replication tool. Whereas EvoCrash was a full clone of EvoSuite (making it hard to update EvoCrash as EvoSuite evolves), Botsing relies on EvoSuite as a (maven) dependency only and provides a framework for various tasks for crash reproduction and, more generally, test case generation. Furthermore, it comes with an extensive test suite, making it easier to extend. This chapter presents the general information about the framework and principal guidelines for contributors.

1.1 Maven modules

Botsing is designed as a framework for crash reproduction and test case generation. The Maven projects is organized as follows:

- `botsing`, the parent module that contains only a `pom.xml` file with the configuration common to all the modules:
  - `botsing-reproduction`, the reproduction engine (see Chapter 4).
  - `botsing-preprocessing`, a tool to pre-process and clean stack traces before calling the reproduction engine (see Chapter 3).
  - `botsing-maven`, the Botsing Maven plugin (see WP4).
  - `botsing-examples`, code examples that could be reused to try Botsing and that are also used for Botsing integration tests.
  - `botsing-commons`, containing Java classes common to multiple sub-modules.
  - `botsing-model-generation`, containing the model generation engine used in model seeding (see Chapter 2).
  - `botsing-parsers`, containing utility classes to parse a stack trace.

1.2 Contributor guide

1.2.1 Building Botsing

To build and install Botsing in your local Maven repository, simply run the following Maven command:

```
mvn install
```

1.2.2 Coding style

The coding style is described in a `checkstyle.xml` file available at the root of the project. The command `mvn checkstyle:check` must succeed for any pull request to be accepted. The rules are:

- No trailing space at the end of the lines.
- No Windows end of line characters.
- Using space and not tabulations for indentation.
- Packages names must start by `eu.stamp.botsing`.
- No unused imports.
- No redundant imports.
- Left curly must be at the end of a line.
- Right curly must be aligned with the code following the code block.
- Code blocks must be enclosed between braces.

1.2.3 Adding dependencies

EvoSuite and other dependencies are managed at the module level. Each module declares a list of Maven dependencies, if you want to add one, simply add it to the list. However, dependency **version** must be declared as a property in the parent `pom.xml` file using the following syntax:

```
<properties>
  ...
  <!-- Dependencies versions -->
  <dependency-artifactId.version>1.1.1</dependency-artifactId.version>
  ...
</properties>
```

And referenced in the dependencies of the module using the following syntax:

```
<dependencies>
  <dependency>
    <groupId>com.groupId</groupId>
    <artifactId>dependency-artifactId</artifactId>
    <version>${dependency-artifactId.version}</version>
  </dependency>
</dependencies>
```

Please check in the list of properties that the dependency version is not already there before adding a new one.

1.2.4 License

Botsing is available under a business friendly license: *Apache-2.0*.

Chapter 2

Botsing model generation

The goal of behavioral model seeding (denoted model seeding hereafter) is to abstract the behavior of the software under test using models and use that abstraction during the search. At the unit test level, each model is a transition system, like in Figure 2.1.

A model represents possible usages of a class (a List in this case): i.e., the possible sequences of method calls observed for objects of that class. For instance, the transition system model in Figure 2.1 has been partially generated from the following call sequences:

```
<size(), remove(), add(Object), size(), size(), size(), get(), remove(), add(
    Object)>
<iterator(),add(Object)>
...
```

Each transition (each arrow Figure 2.1) corresponds to a method call in one of the sequences, and for each sequence, there exists a path in the model. From the model we can derive usages of the List and use those during the unit test generation process each time a List is required during the creation of a test case.

The main steps of our model seeding approach are:

1. the inference of the individual models from the call sequences collected through static analysis performed on the application code, and dynamic analysis of the test cases; and
2. for each model, the selection of usages using the test case generation process.

More detail about model seeding and its evaluation for crash reproduction are available in D3.3 and D3.4.

2.1 User guide

Behavioral models represent the usages of the different objects involved in the project. To learn (i.e., generate) behavioral models for an application, one can use botsing-model-generation that (i) statically analyses the source code of the project and (ii) executes the test cases to capture usages of the objects.

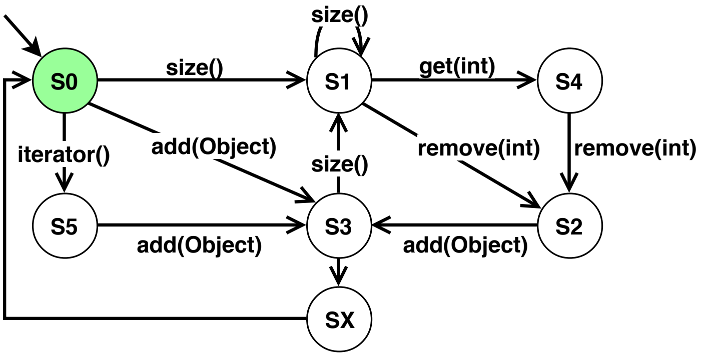

Figure 2.1: Model representing the usages of a list, learned from the static and dynamic analysis of the source code.

2.1.1 Command line interface

The latest version of Botsing model generation in command line (`botsing-model-generation-X-X-X.jar`) is available in GitHub.[1] The models can be generated using the following command line:

```
java -d64 -Xmx10000m -jar botsing-model-generation-X.X.X.jar \
  -project_cp "classpath" \
  -project_prefix "my.package" \
  -out_dir "results/"
```

Where

- `-project_cp "classpath"` provides the class path;
- `-project_prefix "my.package"` tells Botsing to run the test cases that are in the package `my.package` or one of its sub-packages; and
- `-out_dir "results/"` specifies the output directory for the models (models are generated in `results/models`).

Since model generation can consume a lot of memory, we strongly recommend to use the additional JVM options `-d64 -Xmx10000m` when executing the tool.

2.1.2 Maven plugin

See section 4.2.2.

2.2 Contributor guide

This section describes the internal architecture of Botsing model generation.

[1] https://github.com/STAMP-project/botsing/releases

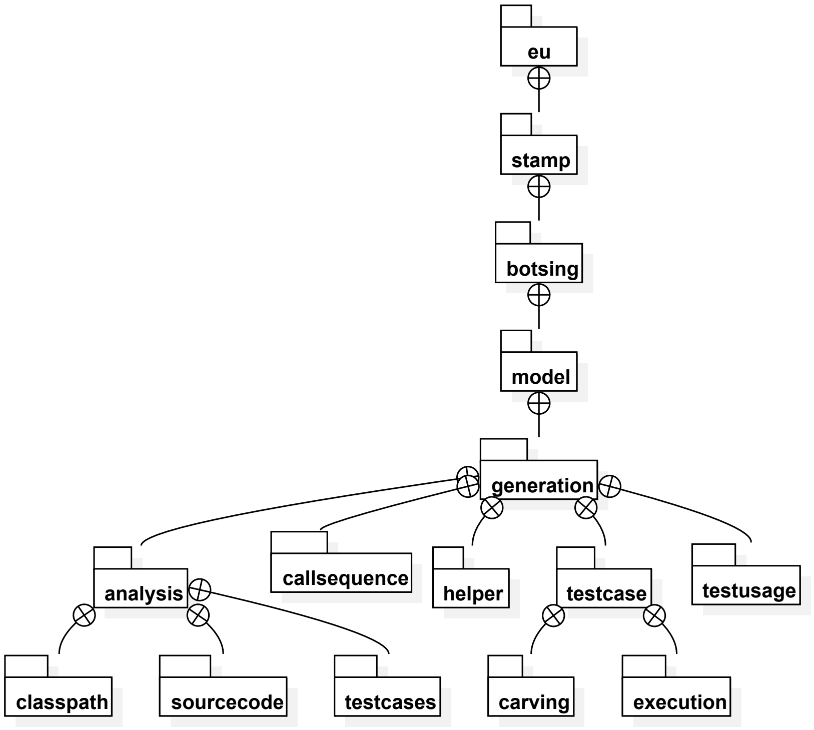

Figure 2.2: Package hierarchy of `botsing-model-generation`

2.2.1 Architecture of Botsing model generation

Figure 2.2 presents the packages diagram of the model generation module of Botsing (`eu.stamp-project:botsing-model-generation`). The main packages are `eu.stamp.botsing.model.generation.analysis` and `eu.stamp.botsing.model.generation.callsequence`. The former applies different analysis to collect call sequences from various resources (*e.g.*, static analysis on source code and dynamic analysis on test classes), and the latter receives the obtained call sequences and use them to generate models using an external library called YAMI.[2]

Class `eu.[...]].callsequence.CallSequenceCollector` contains the method `collect`. This method is the main method of this package and performs static and dynamic analysis . The code of this method is available in the following code:

```
    public void collect(String targetClassIndicator, String outputFolder, List<
        String> involvedObejcts,Boolean isPrefix) {
        [...]
        // Class path handler
        handleClassPath();
        List<String> interestingClasses = detectInterestingClasses(
            targetClassIndicator, isPrefix);
        // Static Analysis
        staticAnalyser.analyse(interestingClasses);
        // Dynamic Analysis
```

[2] https://github.com/xdevroey/yami

```
          dynamicAnalyser.analyse(staticAnalyser.getObjectsTests(), involvedObejcts)
              ;
10        // Storing the object usage of test suites to the output directory
          TestUsagePoolManager.getInstance().savingTestsUsages(Paths.get(
              outputFolder, "carvedTests").toString());
12        [...]
      }
```

This method analyzes classpaths (line 4) and collects the interesting classes for analyze according to the given prefix (line 5). Next, it applies static analysis (line 7) and dynamic analysis (line 9) to collect the call sequences. The collected call sequences are used later for model generation by YAMI (in class `eu.[...].generation.ModelGeneration`).

2.3 Empirical evaluation

In our previous deliverables (D3.4, Section 1.1.2), we presented the results of our empirical evaluation of model seeding applied to search-based crash reproduction. In this section, we complement this evaluation by giving general statistics on the model generated during this evaluation.

Tables 2.1 and 2.2 (resp.) gives information about the number of (resp.) states and transitions in the generated models: the number of models (count), the minimum (min) and maximum values (max), the median, median absolute deviation (MAD), and inter-quartile range (IQR), and mean and standard deviation (sd).

On average, Mockito has the smallest models with a mean number of 7.39 states and 13.66 transitions and a maximum of 40 states and 114 transitions. XWiki has the largest models with an average number of 8.82 states and 18.79 transitions (this number is only toped by Joda-Time with models with 19.18 transitions on average) and a maximum of 376 states and 1627 transitions.

Models are generated from static and dynamic analysis of the source code. Those numbers are consistent with the size of the different projects (see Table 1.1 in D3.4): Mockito is the smallest project with 6.06k $\overline{NCSS}$ and XWiki is the largest project with 177.84k $\overline{NCSS}$.

Table 2.3 provides information about the BFS height of the generated models. The Breadth-First Search (BFS) height is the maximal number of states crossed when traversing the model in breadth-first starting from the initial state to any other state in the model. On average, Commons-math has the smallest BFS-height with 3.09 states, denoting that states in the models are quickly accessible from the initial state. On average, JFreeChart has thee largest BFS-height, with 4.20 to traverse.

project	count	min	median	MAD	IQR	mean	sd	max
Commons-lang	2201	0	6.00	4.45	5.00	8.04	8.96	195
Commons-math	1797	0	5.00	2.97	5.00	7.42	6.08	59
JFreeChart	325	3	5.00	2.97	5.00	8.75	13.11	145
Joda-time	630	0	6.00	2.97	4.00	8.24	12.73	164
Mockito	1176	0	5.00	2.97	4.00	7.39	6.16	40
XWiki	16852	0	5.00	2.97	5.00	8.82	14.82	376

Table 2.1: Number of states per model for the different applications.

project	count	min	median	MAD	IQR	mean	sd	max
Commons-lang	2201	0	9.00	7.41	15.00	17.05	29.24	569
Commons-math	1797	0	7.00	5.93	13.00	15.15	20.58	222
JFreeChart	325	3	8.00	5.93	12.00	17.98	37.36	464
Joda-time	630	0	9.00	7.41	14.00	19.18	49.44	708
Mockito	1176	0	7.00	5.93	9.00	13.66	17.15	114
XWiki	16852	0	7.00	5.93	12.00	18.79	48.72	1627

Table 2.2: Number of transitions per model for the different applications.

project	count	min	median	MAD	IQR	mean	sd	max
Commons-lang	2201	0	2.00	0.00	2.00	3.17	1.85	38
Commons-math	1797	0	3.00	1.48	2.00	3.09	1.54	23
JFreeChart	325	2	3.00	1.48	2.00	4.20	5.42	53
Joda-time	630	0	3.00	1.48	2.00	3.42	3.29	42
Mockito	1176	0	3.00	1.48	2.00	3.38	2.04	16
XWiki	16852	0	3.00	1.48	2.00	3.76	4.35	79

Table 2.3: BFS height per model for the different applications.

Chapter 3

Botsing preprocessing

Botsing preprocessing is an utility tool used to pre-process a piece of log containing an error and extract and optimize a stack trace latter used by the Botsing crash reproduction engine.

3.1 User guide

3.1.1 Command line interface

The latest version of Botsing preprocessing command line tool (`botsing-preprocessing-X-X-X.jar`) is available at https://github.com/STAMP-project/botsing/releases.

Botsing preprocessing has these mandatory parameters (key/value):

- `-i=crash_log` gives the input file path with the stack trace to clean;
- `-o=output_log` gives the path were the output file will be generated .

The actions to perform (clean) in the input log file are:

- `-f` to flatten the stack trace. This action requires the additional `-p=regexp` parameter to indicate which package belong to the software under test.
- `-e` to remove the error message.

Example

To clean the nested stack trace:

```
java -jar botsing-preprocessing.jar\
    -i=crash_log.txt \
    -o=output_log.log \
    -f \
    -p=com.example.*
```

Or to remove the error message in the log file:

```
java -jar botsing-preprocessing.jar \
    -e \
    -i=crash_log.txt \
    -o==output_log.log
```

Note that you can use also both options `-f` and `-e` together.

3.2 Contributor guide

Botsing preprocessing is availabe at https://github.com/STAMP-project/botsing/tree/master/botsing-preprocessing and is part of botsing framework. The main package is `eu.stamp.botsing.preprocessing`, where the class `Main` represents the entry point of the code.

The pre-processing *engine* is represented by the class `StackFlatten`, implementation of the interface `STProcessor`. Each implementation of STProcessor can be an alternative engine that can perform different operations. The operations can be defined by implementing the method `preprocess` and taking into account the two input parameters:

- **lines**, a `List of Strings` containing the stack trace
- **regex**, a `String` representing the package (passed as input parameter).

Chapter 4

Botsing crash reproduction

Figure 4.1 presents an overview of the crash reproduction engine of Botsing. It takes as input a stack trace and the Java binaries required to generate a crash reproducing test case for that stack trace. The initial population is randomly initialized before starting the evolutionary search. If the crash could be reproduced, the engine outputs the test case able to do so.

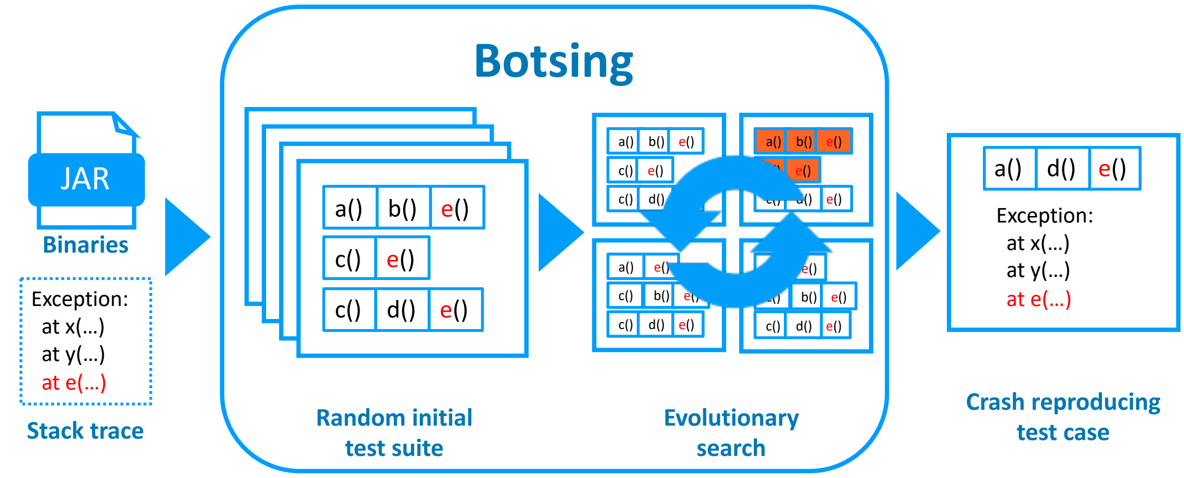

Figure 4.1: Botsing overview for crash reproduction

4.1 User guide

Botsing crash reproduction relies on a search-based to generate a test case able to reproduce a given stack trace. Concretely, this means that, as many other approaches based on meta-heuristics, there is randomness involved in the Botsing process and, despite our best efforts to minimize this effect, two independent runs of Botsing might produce different results depending on the configurations used. This sections provides information on how to use and configure Botsing crash reproduction.

4.1.1 Command line interface

The latest version of Botsing command line (`botsing-reproduction-X-X-X.jar`) is available at `https://github.com/STAMP-project/botsing/releases`. It can be run using the following command line:

```
java -jar botsing-reproduction.jar-X.X.X.jar \
  -crash_log stacktrace.log \
  -target_frame 2 \
  -project_cp "classpath"
```

Where

- `-crash_log stacktrace.log` indicates the file containing the stack trace to reproduce. The stack trace should be clean (no error message) and cannot contain any nested exceptions (see chapter 3 to see how to pre-process the stack trace for Botsing);
- `-target_frame 2` indicates the frame from which the stack trace should be reproduced. This number should be between 1 and the number of frames in the stack trace;
- `-project_cp "classpath"` provides the class path of the application for which a test case should be generated.

Additional parameters can be set. By default, the engine uses the following parameter values:

- `-Dsearch_budget=1800`, a time budget of 30 min. This value can be modified by specifying an additional parameter in format `-Dsearch_budget=60` (here, for 60 seconds).
- `-Dpopulation=100`, a default population with 100 individuals. This value may be modified using `-Dpopulation=10` (here, for 10 individuals).
- `-Dtest_dir=crash-reproduction-tests`, the output directory where the tests will be created (if any test is generated). This value may be modified using `-Dtest_dir=new-outputdir`.
- `-Dmodel_path="path/to/models/"` indicates the path where to find the models. If this option is set, Botsing will use model seeding. Each file should be named after the class it models. See Chapter 2 to see how to generate the models;
- `-Donline_model_seeding=TRUE` if Botsing uses model seeding, this option indicates to generate objects from the models during the search. If this option is set to false, Botsing will initialize a pool of objects (by generating them from the models) during the initialization phase and pick objects only from that pool during the search. This option should be set to FALSE only if there is a large amount of objects required during the search when generating test cases for a given class;
- `-D<property=value>`, allows to configure other parameters of the search.[1]

To check the list of options, use `java -jar botsing-reproduction.jar -help`:

```
$ java -jar botsing-reproduction.jar -help
usage: java -jar botsing-reproduction.jar -crash_log stacktrace.log -target_frame
    2 -projectCP dep1.jar;dep2.jar
  -crash_log <arg>        File with the stack trace
  -D<property=value>      use value for given property
  -help                   Prints this help message.
```

[1] See https://github.com/STAMP-project/botsing/blob/master/botsing-reproduction/src/main/java/eu/stamp/botsing/CrashProperties.java.

```
  -projectCP <arg>      classpath of the project under test and all its
            dependencies
  -target_frame <arg>   Level of the target frame
```

Example

```
java -jar botsing-reproduction.jar -crash_log LANG-1b.log -target_frame 2 -
    projectCP ~/bin
```

4.1.2 Maven plugin

Botsing Maven plugin can be used to reproduce a stack trace in the following way:

```
mvn eu.stamp-project:botsing-maven:botsing -Dcrash_log=<log file> -Dtarget_frame=<
    target frame>
```

Where

- `crash_log` is the parameter to tell Botsing where is the log file to analyze;
- `target_frame` is the parameter to tell Botsing how many lines of the stacktrace to replicate.

In order to reproduce the stack trace, dependencies are fundamental. It is possible to specify them in three alternative ways:

1. from *pom.xml*: i.e. by reading the pom.xml in the current folder and finding the dependencies specified in it. This will be the default behavior if none else has been specified
2. specifying a Maven artifact: Botsing Maven plugin will search in the Maven repository for this artifact and all the dependencies required. An example of the command needed to use this feature is:

   ```
   mvn eu.stamp-project:botsing-maven:botsing -Dcrash_log=ACC-474.log -
       Dmax_target_frame=2 -Dgroup_id=org.apache.commons -Dartifact_id=commons
       -collections4 -Dversion=4.0
   ```

3. from a folder: Botsing Maven plugin will search in the specified folder and gets all the libraries inside it. The command will be something like:

   ```
   mvn eu.stamp-project:botsing-maven:botsing -Dcrash_log=ACC-474/ACC-474.log
       -Dtarget_frame=2 -Dproject_cp=lib
   ```

Concerning the `target_frame` option, it allows to specify how many frames of the stack trace Botsing has to reproduce. Botsing Maven plugin provides three ways to specify it:

1. by using `target_frame` parameter (e.g. -Dtarget_frame=2)
2. by specifying the maximum value (e.g. -Dmax_target_frame=2), in this case it will start from the maximum value provided and decrease it until a reproduction test have been found
3. by reading it from the maximum rows of the stacktrace, in this case no parameter for the target frame should be provided.

4.1.3 Gradle plugin

The documentation of the Gradle plugin for crash reproduction is available in D4.4.

4.2 Contributor guide

This section describes the internal architecture of Botsing reproduction.

4.2.1 Architecture of Botsing reproduction

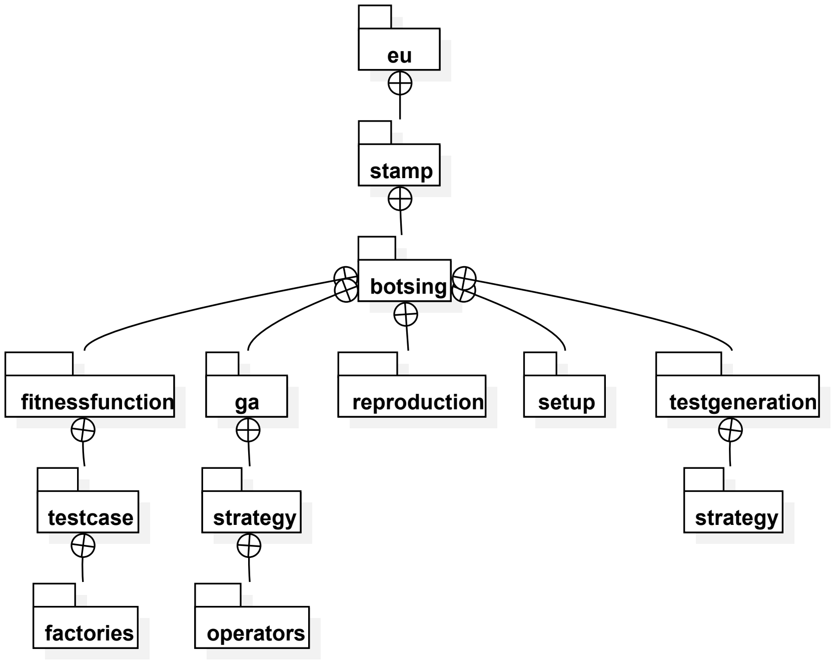

Figure 4.2: Package hierarchy of `botsing-reproduction`

Figure 4.2 presents the packages of the crash reproduction engine (`botsing-reproduction`). The main packages are the `eu.stamp.botsing.ga` and `eu.stamp.botsing.fitness-function` packages. They contain the classes used to execute the guided genetic algorithm and compute the fitness function.

Figure 4.3 details the classes implementing the guided genetic algorithm. A `GeneticAlgorithm` object uses a `TestFitnessFunction` to drive the test case generation process. In Botsing, the `GuidedGeneticAlgorithm` uses a `WeightedSum` fitness function and `Guided-SinglePointCrossover` and `GuidedMutation` operators to perform the crash reproduction search.

The guided genetic algorithm is implemented in the `GuidedGeneticAlgorithm` class by the following method:

```
@Override
public void generateSolution() {
  currentIteration = 0;
```

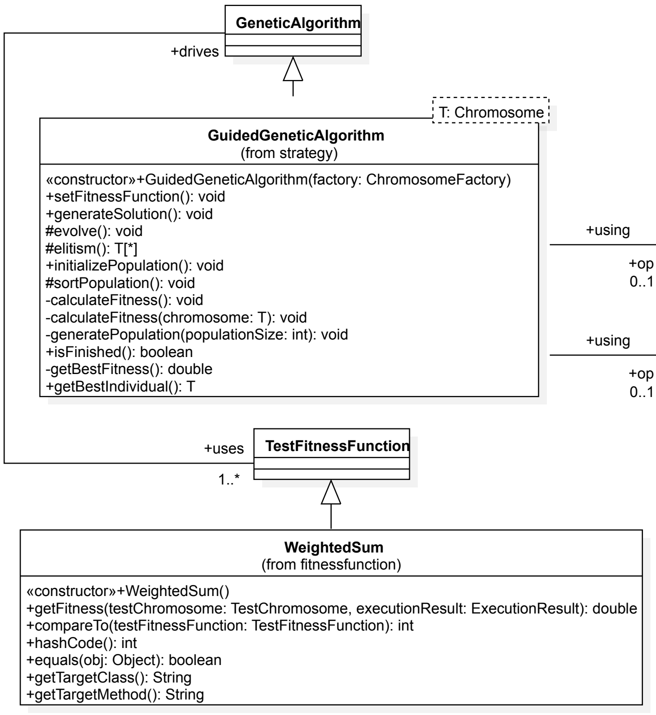
GeneticAlgorithm
+drives
T: Chromosome
GuidedGeneticAlgorithm
(from strategy)
«constructor»+GuidedGeneticAlgorithm(factory: ChromosomeFactory)
+setFitnessFunction(): void
+generateSolution(): void
#evolve(): void
#elitism(): T[*]
+initializePopulation(): void
#sortPopulation(): void
-calculateFitness(): void
-calculateFitness(chromosome: T): void
-generatePopulation(populationSize: int): void
+isFinished(): boolean
-getBestFitness(): double
+getBestIndividual(): T
+using
+op
0..1
+using
+op
0..1
CrossOverFunction
GuidedSinglePointCrossover
(from operators)
+crossOver(parent1: Chromosome, parent2: Chromosome): void
#isValid(chromosome: Chromosome): boolean
-singlePointCrossover(parent1: Chromosome, parent2: Chromosome): void
T: Chromosome
GuidedMutation
(from operators)
+mutateOffspring(offspring: T): void
#doRandomMutation(offspring: Chromosome): void
#insertRandomStatement(chromosome: Chromosome): void
+uses
1..*
TestFitnessFunction
WeightedSum
(from fitnessfunction)
«constructor»+WeightedSum()
+getFitness(testChromosome: TestChromosome, executionResult: ExecutionResult): double
+compareTo(testFitnessFunction: TestFitnessFunction): int
+hashCode(): int
+equals(obj: Object): boolean
+getTargetClass(): String
+getTargetMethod(): String

Figure 4.3: Core classes of `botsing-reproduction`

```
// generate initial population
initializePopulation();

LOG.debug("Starting evolution");
int starvationCounter = 0;
double bestFitness = getBestFitness();
double lastBestFitness = bestFitness;

LOG.debug("Best fitness in the initial population is: {}", bestFitness);
while (!isFinished()){
  // Create next generation
  evolve();
  sortPopulation();

  bestFitness = getBestFitness();
  LOG.debug("New fitness is: {}", bestFitness);

  // Check for starvation
  if (Double.compare(bestFitness, lastBestFitness) == 0) {
    starvationCounter++;
  } else {
    LOG.debug("Reset starvationCounter after {} iterations", starvationCounter);
    starvationCounter = 0;
    lastBestFitness = bestFitness;
  }
  updateSecondaryCriterion(starvationCounter);

  LOG.debug("Current iteration: {}", currentIteration);
33      this.notifyIteration();
      }
35    LOG.debug("Best fitness in the final population is: {}", lastBestFitness);
    }
```

The population is initialized at line 6. The algorithm then loops until the fitness is 0.0 or until the budget is exhausted. The next generation is created at lines 16 and 17. The algorithm has an additional check to prevent starvation at line 23, before starting the next iteration.

4.2.2 Maven plugin

The Maven plugins are available at https://github.com/STAMP-project/botsing/tree/master/botsing-maven. The main classes are available in the package `eu.stamp.botsing`, specifically two Mojos (Maven Old Java Object) and a `ProcessRunner`:

1. `BotsingMojo`, corresponding to *botsing* Maven goal, in *test* scope;
2. `CommonBehaviorMojo`, corresponding to *common-behavior* Maven goal, in *test* scope.

4.3 Empirical evaluation

In this evaluation, we aim to compare the ability of crash reproduction between our two fitness functions: **Weighted Sum** and **Integration Testing**. The former is defined and suggested as the best approach for crash reproduction by Soltani *et al.* [9]. The latter is introduced by us. The definition of this fitness function is available in Section 2.2.2 of the deliverable 3.4 document.

4.3.1 Setup

Case Selection

In total, we performed this evaluation on 124 crashes from JCrashPack [7] on the following projects: `Commons-lang`, `Commons-math`, `JFreeChart`, `Joda-time`, `Mockito`, and `XWiki`.

Configuration Parameteres

We used the same configurations parameters as in our previous empirical evaluation [7].

Infrustructure

We used a cluster (with 28 CPU-cores, 126 GB memory) for our evaluation. We run Botsing with each of the two fitness functions on all of the frames (of the selected 124 crashes), which points to a method in software under test (in total, 951 frames). To address the random nature of the search approaches, we repeated each execution ten times. In total, we executed a total of 19,020 executions for this evaluation.[2]

Data Analysis

Since the crash coverage data is a binary distribution (i.e., a crash is reproduced or not), we use the Odds Ratio (OR) to measure the impact of different fitness functions on the *crash coverage* ratio. A value of OR $>$ 1 for comparing a pair of factors (A, B) indicates that the coverage rate increases when factor A is applied, while a value of OR $<$ 1 indicates the opposite. A value of OR = 1 indicates that

[2] The empirical evaluation infrastructure is openly available at https://github.com/STAMP-project/ExRunner-bash/tree/master/crash-reproduction-no-seeding-integration.

Winner	Crash	Highest Reproduced Frame	Odds Ratio	p-value
Integration	CHART-13b	2	0.00	0.00
	MATH-32b	4	0.00	0.01
	MATH-38b	6	0.00	0.00
	MATH-78b	3	0.00	0.01
	MATH-79b	2	0.00	0.01
	MATH-81b	6	0.00	0.01
	MOCKITO-9b	2	0.00	0.00
	TIME-10b	5	0.00	0.00
	XCOMMONS-928	2	0.00	0.00
	XRENDERING-481	2	0.00	0.00
	XWIKI-14227	2	0.00	0.00
	XWIKI-14475	1	0.00	0.00
	XWIKI-14556	6	0.00	0.00
	XWIKI-14599	1	0.00	0.00
Weighted Sum	MATH-40b	5	Inf	0.00
	MATH-51b	6	Inf	0.00
	MATH-61b	3	Inf	0.00
	MOCKITO-16b	4	Inf	0.00
	TIME-7b	5	Inf	0.00
	XWIKI-12667	6	Inf	0.00
	XWIKI-13196	4	Inf	0.00
	XWIKI-13546	6	Inf	0.00
	LANG-37b	1	Inf	0.00
	LANG-57b	1	Inf	0.00
	MATH-1b	2	Inf	0.00
	MATH-3b	1	Inf	0.00

Table 4.1: Crashes that one of the fitness functions had significantly higher crash reproduction ratio.

there is no difference between A and B. In addition, we use Fisher's exact test, with α=0.05 for Type I errors to assess the significance of the results. A p-value < 0.05 indicates the observed impact on the coverage ratio is statistically significant, while a value of p-value > 0.05 indicates the opposite.

4.3.2 Results

Table 4.1 shows the cases that one of the fitness functions has a higher crash reproduction ratio, significantly. According to this table, we cannot define an absolute winner between these two fitness functions: The weighted sum and the Integration fitness function wins in 12 and 14 cases, respectively. However, according to the collected results, integration fitness function can improve the ability of crash reproduction in botsing. This new fitness function reproduces 18 crashes that the weighted sum cannot reproduce them.

According to the performed manual analysis, in the cases that the weighted sum outperforms against the integration fitness function, the integration fitness function gets stuck in the line coverage of one of the frames, while the weighted sum can pass this line through stack trace similarity. In contrast, the guidance of line coverage of each frame helps the integration fitness function to reproduce some other crashes.

In conclusion, the features in each of the fitness functions (the stack trace similarity in the weighted sum fitness function and the line coverage of all of the frames in the integration fitness function) add different values to the search process, and they should be merged in a fitness function.

Chapter 5

RAMP

EvoSuite is a unit test generator for Java applications (http://www.evosuite.org) originally proposed by Fraser and Arcuri [4]. It implements several evolutionary algorithms and has benefits from many advances over the years. Within WP3, we extended EvoSuite for runtime amplification. Our extension, called RAMP hereafter, includes model seeding for test case generation (presented in D3.3 and D3.4).

5.1 User guide

One of the key functionality of EvoSuite when generating test for a Class Under Test (CUT) is to be able to generate and initialize complex objects used by the CUT. This generation however is random or based solely on the existing test cases by carving (i.e., copy-pasting) objects and methods called on those objects from the existing test cases. With RAMP, we implemented *model seeding* to generate and initialize complex object, based on the behavior described in models (i.e., transition systems) of the classes, and learned from previous executions and static analysis of the system under test. This section shows how to use model seeding with RAMP.

5.1.1 Command line interface

One can run RAMP to generate test cases for a class under test using the following command line:

```
java -d64 -Xmx4000m -jar evosuite-master-1.0.7.jar \
  -class "eu.stamp.ClassUnderTest" \
  -projectCP "$CLASSPATH" \
  -generateMOSuite \
  -Dalgorithm=DynaMOSA \
  -Dsearch_budget=60 \
  -Dseed_clone="0.5" \
  -Donline_model_seeding=TRUE \
  -Dmodel_path="path/to/models/" \
  -Dtest_dir="ramp-tests/" \
  -Dreport_dir="ramp-report/" \
  -Dno_runtime_dependency=true
```

Where

- `-class "eu.stamp.ClassUnderTest"` indicates the class under test for which test cases will be generated. It is also possible to generate a test suite for a whole package using the `-prefix "eu.stamp"` option (see EvoSuite command line documentation for more details);

- `-projectCP "$CLASSPATH"` indicates the class path of the software under test;
- `-generateMOSuite` and `-Dalgorithm=DynaMOSA` indicate to use a multiobjectives evolutionary strategy with the DynaMOSA algorithm;
- `-Dsearch_budget=60` indicates the search budget in seconds allocated for tests generation;
- `-Dseed_clone="0.5"` is the probability to use objects generated from the models. In this case, there is one chance out of two when an object is required to generate it from the models or to randomly generate a new one during the search;
- `-Donline_model_seeding=TRUE` indicates to generate objects from the models during the search. If this option is set to false, RAMP will initialize a pool of objects (by generating them from the models) during the initialization phase and pick objects only from that pool during the search. This option should be set to FALSE only if there is a large amount of objects required during the search when generating test cases for a given class;
- `-Dmodel_path="path/to/models/"` indicates the path where to find the models. Each file should be named after the class it models. See Chapter 2 to see how to generate the models;
- `-Dtest_dir="ramp-tests/"` indicates where the test cases will be generated;
- `-Dreport_dir="ramp-report/"` indicates where the report on the generated tests will be written;
- `-Dno_runtime_dependency=true` indicates to generate plain JUnit test without a dependency to `org.evosuite:evosuite-standalone-runtime`. By default, RAMP generates JUnit tests relying on the EvoSuite framework to, amongst other things, decrease the risk of flakiness.

The generated tests are available in `ramp-tests/` and can be reviewed to improve or correct the oracle (mainly, the assertions) before being added to the test suite of the project.

5.1.2 Maven plugin

The Botsing framework includes a Maven plugin able to generate models for the different classes of a project (see Chapter 2) and generate unit tests using those models. The plugin can be run by using the following commands:

```
mvn eu.stamp-project:botsing-maven:common-behavior \
    -Dproject_cp=$CLASSPATH \
    -Dproject_prefix=my.package \
    -Dout_dir=results \
    -Dclass=eu.stamp.ClassUnderTest \
    -Dsearch_budget=60
```

Where

- `-Dproject_cp=$CLASSPATH` provides the classpath;
- `-Dproject_prefix=my.package` tells Botsing to run the test cases that are in the package `my.package` or one of its sub-packages;

- `-Dout_dir=results` specifies the output directory for the models, the test cases, and the reports;
- `-Dclass=eu.stamp.ClassUnderTest` indicates the class under test;
- `-Dsearch_budget=60` indicates the search budget in seconds allocated to generate tests.

5.2 Contributor guide

In the original implementation of EvoSuite has an object pool which can be used during the test generation. Each object in this pool represents a specific sequence of method calls of certain classes in software under test. When EvoSuite needs an instance of the object, it can either initialize it and call some of its methods randomly, or use one of the existing call sequences in the object pool. In RAMP, we extend the standard implementation of this object pool to fill it using call sequences selected from the given models.

5.2.1 Changes made to the standard EvoSuite implementation

To use model seeding during test case generation in RAMP, we modified three classes:

- `org.evosuite.seeding.ObjectPool`: we added methods to fetch the abstract objects behaviors from models (using the VIBeS[1] tool) and concretize them into executable code.
- `org.evosuite.seeding.ObjectPoolManager`: we implemented a method that receives a class name, and this method checks if the call sequences of this object are available in the pool or not. If it is not available, it will try to find the model of the requested object and add the call sequences from the model to the object pool.
- `org.evosuite.testcase.TestFactory`: we updated the class to use the object pool according to the given model usage probabilities.

5.3 Empirical evaluation

This evaluation aims to assess the effectiveness of RAMP in search-based unit test generation. For this purpose, we compared the scores which are achieved by the generated tests by regular EvoSuite and RAMP in the line coverage, branch coverage, exception coverage, and mutation score.

5.3.1 Setup

Case Selection

In this evaluation, we applied both of the tools on a set of classes from SF110, a well-known search-based unit testing benchmark developed by Fraser *et al.* [5]. This benchmark contains 110 open-source GitHub projects and has been used in many empirical studies to evaluate different search-based software testing approaches [5].

The SF110 bechmark is composed by *Apache Ant* projects. Thus, to build the source and test jars, we provide a compile script[2] that: (1) gets the SF110 sources and tests from the official EvoSuite repository[3] (2) iterates over all projects to build the sources and tests jars using *ant* (3) moves the

[1] `https://github.com/xdevroey/vibes`

[2] Available at `https://github.com/STAMP-project/evosuite-model-seeding-empirical-evaluation/blob/master/compileSF110.sh`.

[3] Available at `http://www.evosuite.org/files/SF110-20130704-src.zip`

binary files of each target software under test to the *bins/benchmark_name* folder required by Botsing to generate models.

Once all projects are compiled, we use Botsing model generation to create models from SF110 binaries. A file[4] defines a list of packages (per project) for which the model generation will be applied. Finally, we apply the botsing model generation script[5] that generates the corresponding models in the */analysis-result* folder. The `classes.csv` file [6] contains the list of classes under tests.

Model Inference

For model inference, we use the botsing model generator. We execute this tool for the selected package of each project in SF110. Each execution of model generation applies a static and dynamic analysis on the source code and test cases of the selected package, respectively.

Configuration Parameteres

We used a budget of 3 minutes for each execution of RAMP and EvoSuite, as suggested by previous studies [5]. We also set the population to 100 individuals. All other configuration parameters are set at their default value [6]. For the search processes (both in EvoSuite and RAMP), we used DynaMOSA that is the best algorithm proposed for unit testing for achieving a higher line, branch, and mutation coverage.

For model seeding, we run each execution with two values for `-Dseed_clone`: 1.0 (*model s. 1.0*), to use the objects generated from the model whenever wee need an object, and 0.5 (*model s. 0.5*), to use the objects generated fromt the model 50% of times and randomly generate objects the other 50% of the time).

Infrustructure

We used a cluster (with 40 CPU-cores, 384 GB memory, and 482 GB hard drive) for our evaluation. We executed EvoSuite (*no s.*) and each configuration of RAMP (model s. 1.0 and model s. 0.5) on each of the 2,593 cases. To address the random nature of the search approaches, we repeated each execution ten times. In total, we executed a total of 77,790 executions for this study.[7]

5.3.2 Results

Figure 5.1 and Table 5.1 present the line, branch and weak mutation coverages of the test suites generated by EvoSuite (*no s.*) and RAMP (*model s. 0.5* and *model s. 1.0*). Line and branch coverage denote the coverage of the source code, while weak mutation coverage denote the ability of a test to expose a defect. Weak mutation considers a mutant as killed if the execution of the test leads to a difference locally observable in the software under test (if the state infection is reached by the test). Contrarily to strong mutation testing, weak mutation testing do not require that change to propagate to the output of the software under test.

From Table 5.1, we do not see any clear difference in the coverage between no seeding and model seeding. To assess the effect size of the difference between coverage achieved by each model seeding configuration and no seeding, we use the Vargha-Delaney $\hat{A}_{12}$ statistic. A value of $\hat{A}_{12} < 0.5$ for a pair of factors *(no s., model s.)* indicates that *no s.* increases the coverage, while a value of $\hat{A}_{12} > 0.5$

[4] Available at https://github.com/STAMP-project/evosuite-model-seeding-empirical-evaluation/blob/master/modelInput.csv.

[5] Available at https://github.com/STAMP-project/evosuite-model-seeding-empirical-evaluation/blob/master/modelGenerator.sh.

[6] Available at https://github.com/STAMP-project/evosuite-model-seeding-empirical-evaluation/blob/master/classes.csv.

[7] The empirical evaluation infrastructure is openly available at https://github.com/STAMP-project/evosuite-model-seeding-empirical-evaluation.

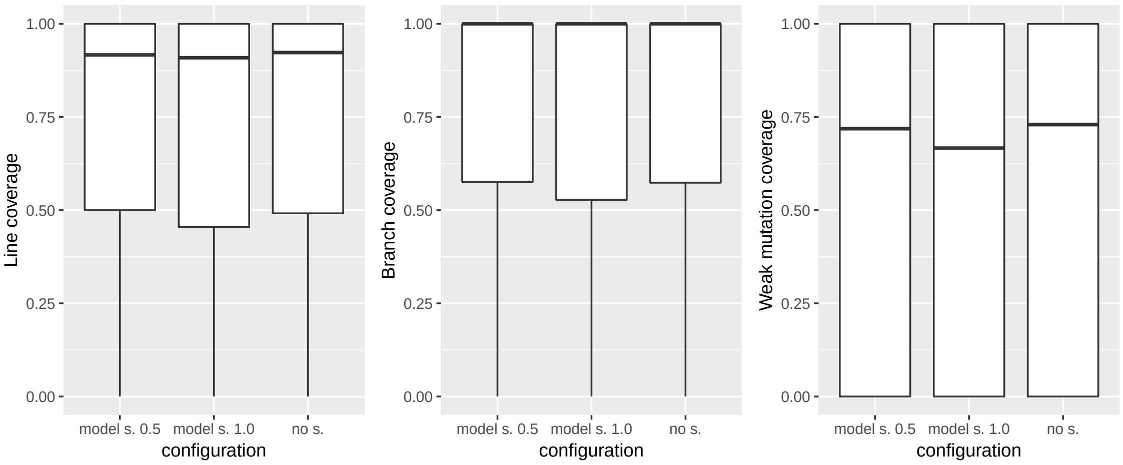

Figure 5.1: Line, branch, and weak mutation coverage of RAMP.

			Line coverage					
configuration	count	min	median	MAD	IQR	mean	sd	max
model s. 0.5	13620	0.00	0.92	0.12	0.50	0.73	0.35	1.00
model s. 1.0	13620	0.00	0.91	0.13	0.55	0.72	0.36	1.00
no s.	13620	0.00	0.92	0.11	0.51	0.73	0.35	1.00
			Branch coverage					
configuration	count	min	median	MAD	IQR	mean	sd	max
model s. 0.5	13620	0.00	1.00	0.00	0.42	0.77	0.34	1.00
model s. 1.0	13620	0.00	1.00	0.00	0.47	0.76	0.35	1.00
no s.	13620	0.00	1.00	0.00	0.43	0.77	0.35	1.00
			Weak mutation coverage					
configuration	count	min	median	MAD	IQR	mean	sd	max
model s. 0.5	13620	0.00	0.72	0.42	1.00	0.54	0.44	1.00
model s. 1.0	13620	0.00	0.67	0.49	1.00	0.53	0.44	1.00
no s.	13620	0.00	0.73	0.40	1.00	0.54	0.45	1.00

Table 5.1: Line, branch and weak mutation coverage of RAMP.

indicates that *model s.* increases the coverage. If $\hat{A}_{12}$ = 0.5, there is no difference between *no s.* and *model s.* on coverage. To check whether the observed impacts are statistically significant, we used the non-parametric Wilcoxon Rank Sum test, with α=0.05 for Type I error. *P*-values smaller than 0.05 indicate that the observed difference in the coverage is statistically significant.

Figure 5.2 presents the effect sizes of the differences between model seeding configurations and no seeding on the line, branch, and weak mutation coverages. For each boxplot, the Figure provides the number of data points, i.e., the number of comparisons between model seeding and no seeding for which the difference was statistically significant (p-value lower than 0.05).

5.3.3 Discussion

On the one hand, from Figure 5.2, we see that, when the difference is significant, model s. 1.0 has a strong impact on the coverage. On the other hand, from Figure 5.1, we see that model seeding 1.0 performs slightly worse than no seeding for line and weak mutation coverage. Practically, this means that model seeding could be activated on demand to increase coverage whenever the search

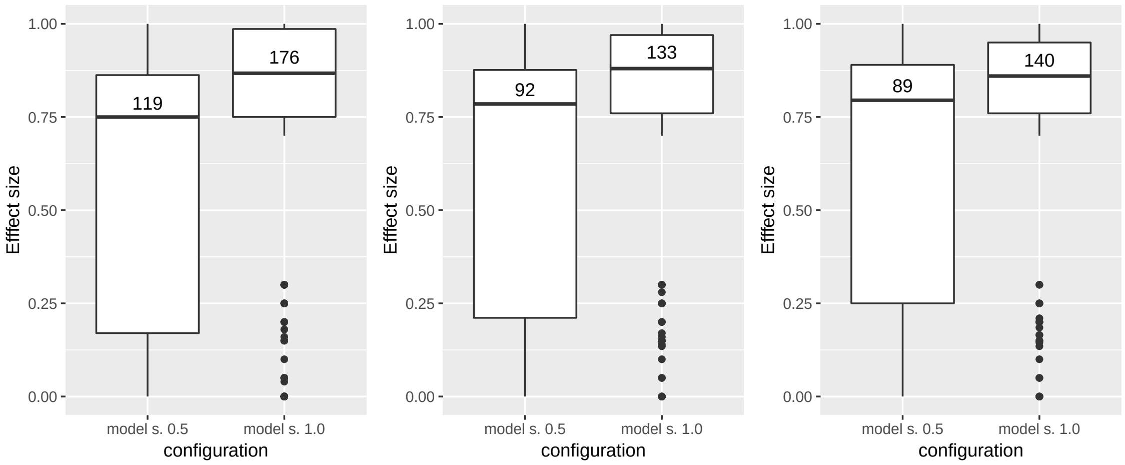

Figure 5.2: Effect sizes of the difference between model seeding configurations and no seeding on the line, branch, and weak mutation coverage.

process does not show any progress for some time. Concretely, if, after a given amount of generations, the search does not show any progress, model seeding should be activated to create objects from the models. Future research include the identification of specific factors influencing the coverage achieved by model seeded search-based test case generation, as well as the improvement of the models themselves.

Chapter 6

Botsing optimization

This chapter contains the descriptions of available strategies to optimize Botsing. Specifically two optimization ways have been found: parallel executions and distributed genetic algorithms. The former was implemented and is currently available in the latest releases, the latter represents a possible way of improvement based on the main characteristics of Botsing architecture (4.2.1).

6.1 Botsing parallel execution

Botsing parallel allows to run an arbitrary number of parallel instances of the Botsing reproduction engine. Specifically the N parallel instances of reproduction engine work on different target frames and, as a certain instance successfully completes the work on a target frame, it start working on the next one (target frame + 1) until the max level of the target frame (set as parameter) is reached. If the work is unsuccessfully completed, it is restarted on the same target frame until a maximal number of attempts is reached.

6.1.1 User guide

Botsing parallel execution needs a specific tool to support Botsing reproduction engine to start *N* parallel instances. The tool requires the following parameters:

- `-project_cp` is the classpath of the project under test and all its dependencies.
- `-crash_log` is the parameter to tell Botsing where the log file to analyze is.
- `-target_frame` is the max level of the target frame. The tool tries to reproduce the crash from the `target_frame` up to the first frame of the stack trace.
- `-N` is the number of reproduction engine instances running in parallel.
- `-X` is the maximal number of times the tool will try to reproduce a frame.

Example

```
java -jar botsing-parallel-jar-with-dependencies.jar -project_cp ~/bin -crash_log
    LANG-20b.log -target_frame 3 -N 2 -X 4
```

All results will be stored in the `test_dir` folder (default value is `crash-reproduction-tests`). This value can be overridden using the additional `-Dtest_dir=new-folder/` parameter. A full example is available at https://github.com/STAMP-project/botsing-parallel.

6.1.2 Contributor guide

The java classes of the module to enable parallel execution of multiple Botsing instances is available in https://github.com/STAMP-project/botsing-parallel. The main package is eu.stamp.botsing, and contains the main class Parallel.

6.2 Distributed genetic algorithms

Botsing searches the best test class to reproduce a stack trace: this search process is based on a genetic algorithm (4.2.1). A Genetic algorithm is a meta-heuristic inspired by natural evolution: it can converge on an solution with certain features after some iteration and a certain time. It is easy to understand that processing times of genetic algorithms can be extremely short and, different executions of the same algorithm on the same data can give different results. In this sense, performance optimization does not means only to optimize the non-functional requirements (e.g. response time), but also to obtain the best possible solution.

6.2.1 Parallel genetic algorithms

A first improvement on simple genetic algorithms is the parallelization of independent executions of the algorithm. Botsing parallel described in Section 6.1 represent the application of this concept.

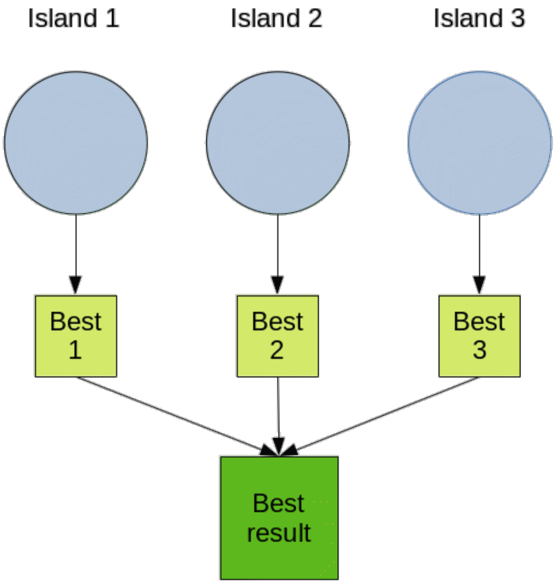

Figure 6.1: Parallel genetic algorithms

Another approach is to design the algorithm to parallelize parts of the the search process. Figure 6.1 illustrates a parallelization approach applied to genetic algorithms following the *island model*. In this configuration, the search process starts by creating a given number of islands. Each island applies a genetic algorithm on the same input data. The results of each island (best individual, in the Figure, *best 1*, *best 2* and *best 3*) are compared against the same fitness function at the end of the search.

Besides the fitness function, genetic algorithms may differ from one another in several aspects:

- creation of new individuals
- number of individuals for each generation

- selection, crossover, and mutation operators
- budget allocated to each algorithm.

All these aspects can be modified, tuning the different genetic algorithms to be applied on the same data with the same fitness function and, potentially, with different results.

Another optimization mechanism is to allow the best candidates of each island (i.e. who obtain the best result against the fitness function) to *migrate* from an island to another.

All these aspects are applicable on a single machine. It is possible to run multiple instances of genetic algorithms (i.e. instances of Botsing) that can autonomously evolve their input data and exchange the best candidates in order to obtain an optimized result.

6.2.2 Distributed genetic algorithms

A further step is the application of the island model of parallel genetic algorithms to separated machines. Distributed genetic algorithms are parallel genetic algorithms running on separated machines.

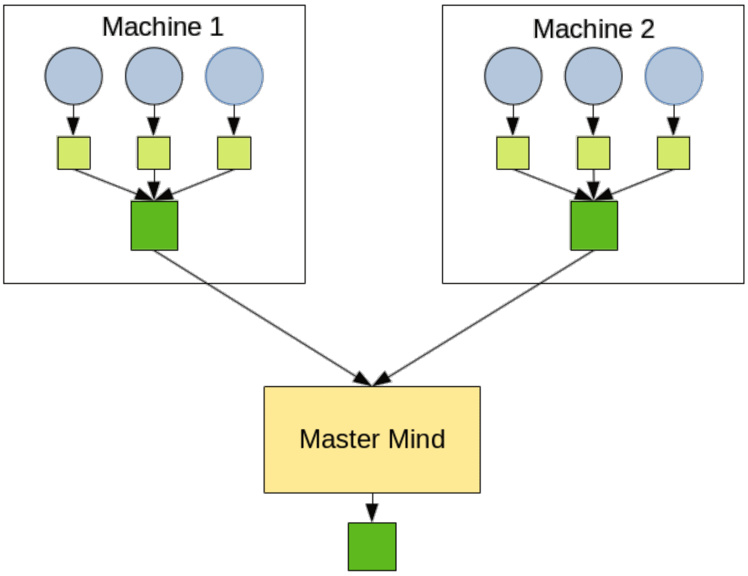

Figure 6.2: Distributed genetic algorithms

Figure 6.2 illustrates the concept. Generally, classical (i.e., non-parallel) genetic algorithms could be run independently on different machines and the best of the best individuals generated by the algorithm is kept. Similarly, parallel genetic algorithms can be run on different machines (as illustrated by Figure 6.2), exploiting the computation power as much as possible.

The figure shows a Master Mind that manages all the cross machines operations, in particular:

- selection of the best individual from all the individuals produced by the machines
- crossover between algorithms on different machines (similar to the crossover between parallel instances).

The Master Mind is also the access point to get the final result(s) of the operation(s). Distributed genetic algorithms can also be applied when the input dataset is wide and cannot be managed on a single machine: in this case each machine work on a section of the whole input dataset.

6.2.3 Potential optimizations of Botsing

The current version of Botsing reproduction allows the user to set various option values for the search algorithm (see Section 4.1). Current implementation of Botsing parallel execution run the same algorithm with the same configuration in parallel processes for the different frames of the stack trace.

From our analysis of distributed genetic algorithms, we identify the potential future optimization for Botsing: the definition of a full-fledged parallel version of the algorithm using the island model, and its evaluation with various configurations, including the topology of the islands, migration, and the configurations of the search on the different islands. Furthermore, the distribution of the algorithm requires the implementation of a Master Mind as be the access point to collect the final result.

Conclusion

This deliverable presents the different tools developed within WP3 and provides for each one the user documentation and developer guide. WP3 focuses on online amplification to leverage information coming from the operations to use them to generate test cases in a DevOps environment.

The Botsing framework includes several tools for crash reproduction and test case generation using runtime information. From a piece of log containing a stack trace, the stack trace preprocessor documented in chapter 3 can extract and optimize the stack trace to maximize its reproduction chances by the crash reproduction tool documented in chapter 4. This reproduction can be further enhanced using the behavioral models generated by the model generator described in chapter 2. Those models can also be used in RAMP, described in chapter 5, to generate unit tests. Finally, the running time of the crash reproduction engine can be optimized using a parallelized implementation described in chapter 6.

Progress beyond state of the art

STAMP aims at extending existing online testing techniques leveraging log files that are commonly used in both commercial or open source projects [1] to log important events such as error or warning messages, as well as some historic information generated during normal execution. First, we used log data for deriving common patterns in logs and for automatically learning labelled transition systems mirroring common usages of objects [3]. Second, online test amplification has been achieved by generating test cases using the derived labelled transition systems as seeds (see chapter 5). As such, we were able to replicate anomalies and software crashes reported in log data [7,8,10]. Finally, the quality of the information that is available on the running system during search-based test case generation or crash reproduction impacts the quality of the generated tests: too detailed information could degrade system performance (and the exploration of the search), while too general log data may prevent event replication (by degrading the exploitation of potential information). This balance between exploration and exploitation is a well-known problem in search-based software engineering. Therefore, we went beyond traditional search-based unit test generation techniques by proposing a new code instrumentation approach, that improves the testability of the code and allows to automatically generate class integration tests [2] and enhance search-based crash reproduction (see chapter 4).

Bibliography

[1] J. Candido, M. Aniche, and A. van Deursen. Contemporary Software Monitoring: A Systematic Literature Review. *Submitted to the IEEE Transactions on Software Engineering*, 2019. (under review at the time of writing this deliverable).

[2] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen. Towards Integration-Level Test Case Generation Using Call Site Information. In *Submitted to the 42nd International Conference on Software Engineering*. ACM/IEEE, 2019. (under review at the time of writing this deliverable).

[3] P. Derakhshanfar, X. Devroey, G. Perrouin, A. Zaidman, and A. van Deursen. Search-based Crash Reproduction using Behavioral Model Seeding. *Submitted to the Software Testing Verification and Reliability journal*, 2019. (under review at the time of writing this deliverable).

[4] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM Press, 2011.

[5] G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):8:1–8:42, Dec. 2014.

[6] J. M. Rojas, G. Fraser, and A. Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, aug 2016.

[7] M. Soltani, P. Derakhshanfar, X. Devroey, and A. van Deursen. A benchmark-based evaluation of search-based crash reproduction. *Empirical Software Engineering*, (731529), aug 2019.

[8] M. Soltani, P. Derakhshanfar, A. Panichella, X. Devroey, A. Zaidman, and A. van Deursen. Single-objective versus Multi-Objectivized Optimization for Evolutionary Crash Reproduction. In *Proceedings of the 10th Symposium on Search-Based Software Engineering (SSBSE '18)*. Springer, 2018.

[9] M. Soltani, A. Panichella, and A. van Deursen. A Guided Genetic Algorithm for Automated Crash Reproduction. In *International Conference on Software Engineering (ICSE)*, pages 209–220. IEEE, may 2017.

[10] M. Soltani, A. Panichella, and A. van Deursen. Search-Based Crash Reproduction and Its Impact on Debugging. *Software Engineering, IEEE Transactions on*, 2018.