Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master's Thesis

Towards History-Guided Generation of Tests with Maximum Impact

submitted by
Elias Thomas Hartz

submitted
February 2nd, 2015

Supervisors

Alessandra Gorla
Juan Pablo Galeotti

Reviewers

Prof. Dr. Andreas Zeller
Prof. Dr. Sebastian Hack

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, den _____ _____

(Datum/Date) (Unterschrift/Signature)

Contents

1 Introduction

The perfect software developer does not exist and as such any software project will contain defects at some point. Failures caused by such errors, even by those which are seemingly small in scope, can have disastrous results [16]. Through *software testing*, one tries to achieve some level of confidence on the project's behavior [43], implementing a mechanism to detect if the software is generally doing what it is supposed to do. Software-based testing frameworks have been widely adopted by the industry and their introduction has had a considerable impact on the efficiency and the costs of the software development process as well as on software maintenance ([9], [43], [55], [47]). By utilizing testsuites to automatically and systematically check a system's functionality and ability to integrate with other projects, a great deal of time and effort can be saved. However, actually creating the required testsuites is among the most time-consuming tasks to be found in software development, which can demand up to 50 percent ([2], [43]) of the resources available over the course of development.
Consider e.g. *unit testing*: In this paradigm, one attempts to test the individual components [36] which as a whole make up the entirety of the project. Each is tested in isolation, meaning that one needs to ensure correct behavior of all functionality in all states the component may be in without integrating it into the larger-scale system. This process includes detecting and covering rare corner cases or states that might never be reached during a typical execution of the program. All of these cases, along with scaffolding functionality to allow for isolated testing, must be implemented, if a meaningful testsuite is to be obtained. For larger projects, this is an expensive and time-consuming undertaking [55], not to mention that it can be quite challenging depending on the complexity of the project under test. With this in mind, the benefits of improving upon the software testing process are obvious and as a result various research topics have emerged, among it the field of *automated test generation*. However since Howden [26] showed in 1976 that there is no computable criterion that would ensure the replayability and validity of a test and a project's specification is rarely in a format from which a program can derive test cases, no automatic test generator can fully relieve the programmer from his or her responsibility.

1.1 Topic of this Thesis and Motivation

One particular class of automated test generation techniques is known as *white-box test generation*, its main advantage being that it works without any form of semantic specification or annotation of the project [2]. It enables the developer to obtain test

cases from the pure source code but as a result all generated test cases will capture the current behavior of the code as-is, including any defects it might contain. If executed on the code it was derived from, such a testsuite will always report that no failures were encountered. The benefits of automatically generated tests lie in lessening the required costs and time, in contrast to manually implementing numerous test cases, the developer's task is reduced to verifying the generated tests.
However, a study performed by Fraser et al. [19] points out that the quality of software testing will not necessarily improve once test generation has been automated, and only inspection and correction of these tests is performed manually. In particular for complex classes, understanding the generated test and its assertions seems to be challenging for developers. According to the authors, the quality of a generated test is not only a matter of its adequacy against e.g. structural criteria like statement- or path-coverage but furthermore depends on factors that are much more difficult to quantify, e.g. how easy to understand a generated test case is and if it helps the developer to come to the right conclusions.

These findings motivate this master thesis. We aim to take advantage of a software project's history to direct the generation of tests, focusing the efforts of the generator on sections of the code that are behaving differently. The long term goal is to enable automatic test generation to place an emphasis on the modified behavior of the software, stressing the semantic impact of the tested code modifications. This thesis should be seen as a first step towards that goal.
Combining data from a static syntax change analysis between different versions of a program with execution traces from these versions, the main contribution of our work is the research tool *jHisteg* which is capable of approximating the impact of source code modifications between different versions of a program. Outputting an ordered list of targets for an automated test generator, *jHisteg* can be used to guide test generation towards those sections of the code that have shown the gravest behavioral changes.

1.2 Structure of this Thesis

In chapter 2 we present related work and categorize this contribution, putting it into context. Chapter 3 illustrates our approach and elaborates on the design decisions that have been made. Its realization as a Java-based tool [29] called *jHisteg* is detailed in chapter 4, an empirical study on its capabilities is presented in chapter 5. Finally, chapter 6 summarizes the previous chapters' results and draws conclusions from them, as well as suggesting what steps should be taken next in future work.

2 Related work

In 1975, Goodenough and Gerhart [24] introduced testing adequacy as the central problem for test generation and Miller and Melton [41] proposed the automatic generation of data for testing purposes as a first step. One year later, Howden [26] showed that no computable criterion could ensure replayability and validity. In fact, Weyuker and Ostrand [58] showed later that they are exclusive, which caused a shift to judging testing adequacy criteria on their practicality and popularized structural criteria such as code coverage, proposed as early as 1963 by Miller and Melton [42]. Numerous adequacy criteria and different techniques to systematically and automatically generate tests have been proposed over the years [61].

One of the first test generation tools was developed by Csallner and Smaragdakis [14], their *JCrasher* attempts, as the name would suggest, to generate executions through random testing that lead to a crash of the program under test. Though tools like *Randoop* by Pacheco and Ernst [46] were more sophisticated, capable of e.g. directing the test generation with a previous execution's result, they required the user to manually annotate the source code to function optimally. As such, *JCrasher* can be considered the first fully automated test generator. In 2005, Godefroid et al. [23] proposed to use King's dynamic symbolic execution technique [35] to the maximize code coverage of a generated testsuite. Their work resulted in the *DART*-tool which was capable of generating high-coverage tests fully automatically without any manual annotation or direction on any compiling program. This caused a paradigm shift towards generating tests with the goal of systematically maximizing a specific adequacy criterion. For instance, Sen et al. [52] introduced a technique to approximate pointer constraints, enabling test generation to systematically cover complex dynamic data structures in order to achieve e.g. high coverage in those code sections as well.
In 2011, *EvoSuite* was introduced by Fraser and Arcuri [18]. Using a search-based genetic algorithm, this tool is capable of generating whole testsuites along oracles in the form of effective assertions. This novel approach was picked up and extended multiple times, inter alia through an integration of dynamic symbolic execution by Galeotti et al. [20]. Vivanti et al. [57] incorporated data-flow criteria into *EvoSuite* and Shamshiri et al. [53] proposed an improvement to better propagate occurring errors to code sections where they can actually be observed and detected. It is evident that Fraser and Arcuri's work was highly influential. As this thesis primarily deals with obtaining usable data on what modifications occurred during two versions to guide test generation, no custom test generator of our own has been contributed. However, with powerful tools like *EvoSuite* employing customizable fitness functions, we can argue that leveraging impact-based data is possible today.

With the advent of robust automated test generators came an integration into real-life software projects, for example by Jin et al. [56] at Microsoft Research: Employing their *Pex*-tool to generate tests for an "*extremely well tested core .NET component*", they obtained encouraging results, with the testsuite even uncovering a serious, previously unknown issue. Recent proposals of software development models leveraging the benefits of automated test generation further continue this trend: Introduced by Campos et al. [15], *continuous test generation* combines automated testing with the continuous integration practice by Booch [11], giving a muster of how to make use of a project's development history in order to make the most of a limited testing budget. Akin to our thesis, Campos et al. suggest to focus on new and modified code sections in general, the distinction is that we take the actual code modifications into account in greater detail.
All such integrations of automated test generation into a development process face the question when to test something. Consider e.g. a regression testing context: What exactly has to be tested in a new version, what was affected? This problem is known as *test selection* or, in case one also provides new test cases, *test augmentation*. In 1997, Rothermel and Harrold [49] proposed a technique based on control-flow graph traversal for automatically selecting test cases that executed code with modified behavior with respect to a previous version, with Harrold et al. [25] later expanding on this using *Java Interclass Graphs*, to take class hierarchy and invocation-relationships into account. In 2006, Apiwattanapong et al. [6] presented the *Matrix* which combined known static techniques like data- and control-flow analysis with dynamic symbolic execution to determine the adequacy of a test suite against a modified code-base. Santelices et al. [50] later performed an evaluation of an improved version of *Matrix* on larger, real-life software projects, hence validating the practicality of the technique. Further improvements were made by Xu and Rothermel [60] who enabled the generation of tests for previously uncovered sections of the code and Taneja et al. [54] who optimized the augmentation process by pruning irrelevant paths.

Before one can augment a testsuite, one must determine what has actually been changed, Bohner and Arnold [10] call this *software change impact analysis*. A formal model for program dependencies was proposed by Podgurski and Clarke [48] and a technique by Agrawal and Horgan [1] known as *dynamic program slicing* has arisen as one of the most popular approaches. Law and Rothermel [37] proposed a technique based solely on execution traces in 2003 while Schuler and Zeller [51] used a metric-based approach using mainly source code coverage when developing their tool *Javalanche*. This thesis follows in their steps, using execution-derived data as well.
As such there are also thematic similarities to a recent publication of Marinescu et al. [39] who developed a framework for extracting and executing individual versions from software repositories with the goal of analyzing how criteria like coverage change over development history. While their framework was not used in our implementation, they have made important remarks about how to set up an empirical study requiring executions of different program versions. Speaking of this thesis' technical realization, the work of Fluri et al. [17] is of high relevance as their *ChangeDistiller*-tool is used

to compute syntactical changes between two versions, data that is then combined with trace-derived information.
Finally, of particular importance to this thesis is a study from 2013, in which Fraser et al. [19] used *Evosuite* to explore the practical usefulness of automatically generated tests and found, as stated previously, that using automatic test generation can in fact even be detrimental to the software's quality. The authors point out that we need to generate easier-to-understand tests if we are to effectively make use of automated test generation in real software development processes. This result is the main motivation behind this thesis' conception, with our long-term goal being to alleviate the challenges currently faced by users.

3 Generating Tests with Maximum Impact

Software development is an incremental process in which previous versions of a project are used to construct a new iteration, which typically contains refinements or additional functionality. Syntactic modifications of the source code and the resulting differences in semantics are a part of this process. If one now assumes that the previous version of a project is well tested and contains no grave defects, intuition tells us that it suffices to test what has been introduced during this development iteration, which comes down to limiting test generation to those parts of the source code which have been modified during the current development phase. To that end, we need to identify the full impact of changes made by the programmer. Theoretically, testing only what behaves differently would suffice after each iteration on the code, since for the unmodified parts our previous testing results still hold, provided you can identify which semantics changed and which stayed the same with perfect accuracy of course. Apiwattanapong et al. [6] call this *test augmentation*.
In the context of this thesis, we do not intend to evolve test cases from version to version, we aim to specifically create tests for each of the changes the developer actually implements. The idea is that the resulting test case is smaller in scope and on a level appropriate for the change in question, e.g. not testing an entire class if only a single method was affected. In addition to this, the approach proposed here should enable a much closer connection between the test and the actual software development process once automatic test generation can make full use of the data extracted from the project history, since what is being tested is based on what has been modified. Furthermore, test generation can be directed towards those parts of the code where the semantics have been altered the most, confronting the developer with the gravest effects of his or her changes first. In the long-term, this should help to alleviate the issues found by Fraser et al. [19] and help to make automated test generation more viable in practice.

This thesis deals with obtaining usable data from the projects history to guide test generation, as such it focuses on approximating the impact of source code changes on a whole project. The goal is to provide means to guide test generation towards those modifications that caused the most impact.

3.1 Requirements

Our notion of a project's history is based on versions of that project. A version is a fully-featured, runnable snapshot of the project, capturing its exact state at a single point in development time. When using a modern version control system, this corresponds to a particular commit or revision of that software repository (see chapter 4). These versions must be independently stored and accessible, as we are going to compare both their source code but also dynamically generated results obtained through executing the respective version. For the examples in this chapter, we will assume that we have access to both raw sources and successfully compiled sources. Aside from these requirements, executing a project in some way also necessitates that we have information about its general structure, e.g. where its sources are or where a testsuite is located but such technical requirements will not be of further interest until chapter 4. It is worth to note that our approach requires only the project itself, no form of external annotation, configuration or manual guidance by a human is needed.

3.2 Change Detection

If we want to systematically cover only the sections of a program under test that have changed, we will need to identify these sections first. For this thesis, we distinguish between the notion of *syntactic modifications* and *changes in code semantics* extracted from actual program executions.

3.2.1 Syntactic Source Code Changes

At first glance, a syntax change is easy to define. Any modification of the source code between two versions, that is any addition, replacement or deletion of a language construct belonging to the programming language being used, can be seen as a syntactic change of the program in question. Since modern software projects typically use a version control system, we can quickly obtain information about which files were modified by the programmer. If that modification matches above definition, we have a syntax change of the source code.

For the vast majority of programming languages, source code can be represented using an *abstract syntax tree* or *AST* for short, a data structure modeling the syntactic arrangement of a particular program which is typically enriched with additional semantic information by the compiler e.g. linking a method's invocation to that method's defining AST [59]. Since the different language constructs are all represented as is in the AST, a comparison of two ASTs will result in syntax changes as defined above.
It is important to stress however, that while we can easily locate a syntax change it is difficult to infer the severity of said change from the AST. Any given modification on the syntactic level could potentially have grave effects whose impacts can be felt over the entire program. Consider for example changing the value of a constant which typically

involves only a few characters since you simply set it to a different value. This may in turn alter the result of many computations performed at different places in the code, which might lead to other branches being taken, different objects being created and data structures ending up in a vastly different state. Despite the syntactic change being a seemingly tiny one, its effects must be considered not only locally but on a global scope, at least for most object-oriented languages. For imperative ones, this may not be the case as e.g. Milanova et al. [40] show that for C, the complete call-graph can be obtained using only syntax information.
On the other hand, even numerous syntax changes may have no effect on the program execution at all, failing to change the code's semantics: First and foremost, there are parts of most programming languages that we can simply ignore for our intents and purposes, most notably comments as they have no semantic value for the execution of the program (and are eliminated during compilation anyway). In addition to such constructs, we might also encounter modified source code with identical observable behavior nonetheless. In the context of mutation testing, a technique performing syntax changes on programs under test at random, a syntax change without an effect on code semantics is known as an *equivalent mutant* [51].
These equivalent mutants, in our case equivalent versions of a selected section of the code, represent a challenge since they must be detected before performing subsequent analysis steps, which are completely superfluous in this case. Recommending to retest sections of the code which behave identical in all regards is factually wrong. In general, the problem of equivalent mutants is undecidable [12] though heuristic approaches have been developed and yielded good results [8] [44] [51]. Due to the problem of equivalent mutants and the resulting imprecisions, we have to combine our information about syntactic changes between versions with other metrics.
Figure 3.1 shows four different programs with each having identical observable behavior despite syntax modifications, with one even making some drastic changes to the program's control flow. Their output visible to the user is the same in spite of these modifications.

3.2.2 Semantic Source Code Changes

In the context of this thesis, a modification of the source code's semantics is an actual observed difference in the program's execution, thus our notion of a semantic change is a dynamic one if you will. To this end, we are comparing an execution trace obtained from one version of the source with a second trace from another version, matching one concrete call of a method onto another. If the execution differed by e.g. taking a different branch, we say that our trace has diverged from the one we compared it to. This divergence is the semantic effect of a syntactical change earlier in the trace, typically inside a method directly or indirectly invoking the one with the divergence. For instance, the programmer could have modified a computational statement assigning a value to a local variable. If this variable is then passed as a parameter to another invoked method, that method could behave differently due to this new value, e.g. by taking a different branch or executing a loop more often.

```java
/** Example program */
public class EntireProgram {

  public static void main(String[] s){
    int i = 5;
    while (i >= 0){
      if (i == 0)
        System.out.println(i+"!!!");
      else
        System.out.println(i+"...");
      i = i - 1;
    }
    return;
  }




}
```

(a) Original Java program

```java
/** Example program */
public class EntireProgram {

  private static boolean bool = false;

  public static void main(String[] s){
    int i = 5;
    while (i >= 0){
      if (i == 0 || bool){
        System.out.println(i+"!!!");
        bool = true;
      }
      else
        System.out.println(i+"...");
      i = i - 1;
    }
    return;
  }

}
```

(b) Program with multiple syntax changes related to new field named *bool*

```java
/** Example program */
public class EntireProgram {

  /** Program entry point */
  public static void main(String[] s){
    int i = 5;
    // Counts down from 5 to 0!
    while (i >= 0){
      if (i == 0)
        System.out.println(i+"!!!");
      else
        System.out.println(i+"...");
      i = i - 1; //decrease counter
    }
    return;
  }

}
```

(c) Program with additional comments

```java
/** Example program */
public class EntireProgram {


  public static void main(String[] s){
    int i = 5;
    while (i > 0){
        System.out.println(i+"...");
      if (i == 1)
          System.out.println("0!!!");
      i = i - 1;
    }
    return;
  }

}
```

(d) Program with numerous syntax changes affecting branch conditions and control flow structure

Figure 3.1: Examples of equivalent code sections with syntax changes

We introduce the concept of a semantic change for two main reasons, the first being the already mentioned equivalent code sections: Despite being modified by the programmer, a section of code still behaves identically to the unaltered version. While it is by no means a perfectly reliable metric, taking exactly the same path through a method can indicate equivalent code semantics. Schuler and Zeller [51] obtained good results using a similar technique to rank modifications produced through mutation testing with the goal of detecting equivalent mutants.
The other main motivation behind this concept is the fact that a syntax change and its actual semantic effect are not necessarily close together but we are precisely interested in this effect or impact. Passing a different constant value as parameter to a method for example involves a syntactic change at the invocation statement but the actual effect of this modified value, if any, can only be observed inside the invoked method. That method may very well reside in a syntactically unmodified class, hence the semantic effect of the syntactic change is a full method invocation away.

3.3 Impact Analysis

Our overall task is to analyze the effects of syntactic changes on the program's semantics and to order these syntax modifications by magnitude of their impact. To this end, we have introduced the concept of syntactic and semantic changes between two versions of our project. Assessing the effects of a syntax change on the semantics is made possible by comparing trace execution data, which yields exactly those semantic effects as observed during the execution. If two traces derived by similar means diverge in code sections without syntax changes, then the programmer has made modifications that caused a different behavior on at least one occasion.

To compare execution traces and obtain meaningful results from that comparison, we require a suitable level of abstraction. After all with the memory layout being different each time we start our program, we would detect e.g. completely different objects if we would attempt to compare by reference. We are more interested in the paths taken through methods as well as how we enter and leave it, since a method typically implements some form of encapsulated functionality computing a result from some inputs. Furthermore, we must assume that our subject traces are actually matchable onto another, that is they are generally using the same functionality for the very same purpose. The reason behind this is that we are performing our trace comparison because we are interested in the impacts of syntax modification and we can make the most accurate statements about its effects if the changed section is used in similar fashion before and after applying the change.
To come by an approximate representation of what happens inside our program, we decided to record which statements are executed and how often, as well as what kind of objects are being passed around as parameters and return values between methods. This provides us with the execution's exact paths while abstracting from the precise internals

of the computation. Tracking the value of the arguments gives us some intuition on why a control-flow change might occur and recording the returned result gives us some measure on how strong those changes have impacted the overall method behavior.
Making sure that our traces are matchable on the other hand is more of a question on how we generate them. Using means that are present and unmodified in both versions satisfies this requirement, ideally we use a present testsuite which ensures that our traces both share the same purpose and start at the same entry point.

Having defined an adequate abstraction and means of obtaining suitable traces, it is left to discuss on the exact nature of a trace divergence. Any observable difference between two traces in our abstraction is called a trace divergence. As such one method invocation can diverge from another in up to three ways, either by getting called with different parameter values, by returning another value or by execution of different statements, in other words by having an execution path unlike the other. Everything else is abstracted. If now two traces diverge, that implies that the method in question has now behaved differently than before. Ultimately this may mean that a different value is being returned or the state of an object may differ, which in turn can have grave ramifications on the rest of the program. By construction, we are aware of the call chain and can thus determine which syntactically modified sections of the code where executed before any given trace divergence, which we deem responsible for the divergence.
It is fundamental to be aware of the fact that a divergence does not signal the presence of a defect inside the source code in any way. We can only deduce that a concrete execution is different for our two versions, hence a modification had an effect. We will use this information to assess the impact of a syntax modification made by the programmer in the next step. Figure 3.2 contains a detailed example that illustrates the impact of syntactic changes on program semantics in greater detail.

By mapping the observed semantic differences to the responsible syntax modification, we can approximate the size of the impact of said modification. If for one syntax change the average the number of divergences over all traces his higher than for another, then it had a graver impact. If a modification only causes a few divergences inside the method and a different result which leads to no further observable differences in our abstraction, its effect was likely only local. Plainly speaking, the more effects we can observe for a change, the higher is its gravity and overall impact. Chapter 4 contains information about the exact metrics that are used for impact analysis.
Note that any implementation that compares execution traces as described will of course also detect divergences in code sections that were syntactically modified. We must be very careful not to draw false conclusions from this result, after all a typical syntax change adds, removes or modifies instructions of the code. As such, the executed statements differ, the trace obviously differs as well and we report a trace divergence. For our means, those divergences are negligible since we focus on unmodified sections for impact analysis. However, recall the possibility of having a syntactic change but no difference semantic. If a syntax modification causes no or only a minor number of different statements to be executed, causes no divergences in invoked methods and has no effects on

```
1  public class Casino {
2     /** change of winning between [0f,1f] */
3     public static final float c = 0.05f;
4     public static final int bet = 250;
5     public static final int win = 750;
6     public static final int seed = 13;
7     public static final int kick = 99;
8
9    public static void newGambler(){
10     int amountOfMoney = 15000;
11     int howMuchToSave = 5000;
12     new Gambler(
13         amountOfMoney,
14         howMuchToSave
15       ).gamble();
16   }
17 }
18
19 class Gambler {
20     private int m;
21     private final int l;
22     private int kickoutTimer = Casino.kick;
23
24   public Gambler(int money, int limit){
25     this.m = money;
26     this.l = limit;
27   }
28
29   public void gamble(){
30     Random r = new Random(Casino.seed);
31     while (kickoutTimer!=0){
32       if (Conscience.willWinBack(m, l))
33         if (r.nextFloat()>= Casino.c)
34           m += (Casino.win-Casino.bet);
35         else m -= Casino.bet;
36       else break; //leave loop
37       --kickoutTimer;
38     }
39     return;
40   }
41 }
42
43 class Conscience {
44   public static boolean willWinBack(
45       int money, int limit){
46     return money >= limit;
47   }
48 }
```

This program always terminates, either because the `willWinBack`-method of `Conscience` returns `false` or the `kickoutTimer` of the `Gambler` instance reaches 0 and the loop is left.
Any change to the constants in `Casino` has the potential to produce a trace that diverges in either the `Gambler` or the `Conscience` class although our syntactic modifications take place exclusively in the `Casino` class.
Now where would we first identify an execution trace divergence when comparing the original and modified traces? If the `seed` is changed (and the pseudo-random generator produces sufficiently diverse results) we will first observe an execution of line 35 instead of 36 or vice versa. Any change to the other constants leads to either a different result being returned in line 47 in the trace of the modified version or a different number of loop iterations, meaning one trace would again execute line 32 while to other would execute the `return` in line 40 next. Note that we would not treat a different value of `m` in line 35 as a diverging trace section, as concrete values are only stored for parameters and return values in our abstraction. We only record which statements are being executed!
The traces might also be identical despite the presence of a syntax change in `Casino`: For example if the chosen random seed produces only loses for the gambler until one of the two termination condition is valid, the value of `win` is never used and can therefore be set to any integer number without impacting the trace (since line 35 would essentially be dead code in this scenario).

Figure 3.2: Connection between syntactic and semantic changes

the method's results, this might signal that the syntax change had no effect. If nothing has changed then the chosen technique for generating the trace is unable to cover your modification at the very least, which in the case of a testsuite means that your new code is not covered.

Through combining our statically computed data about syntactic changes with differences in execution obtained through an adequate abstraction, we receive for each syntax modification both a metric describing its impact as well as concrete, observed differences. We propose that this information is leveraged to generate a specialized test case that mimics the programmer's changes, presenting him or her with their effects of which some might be unintended or even undesired.

3.4 Integration into the Software Development Process

Testing large-scale software projects is an expensive task, demanding a considerable amount of time and effort [55], hence reducing these costs through automated test generation can only benefit the development process. When dealing with a continuous integration context as proposed by Booch [11] and a fixed budget for testing, Campos et al. [15] discuss how to allocate your limited resources based on historical data. They propose to prioritize newer classes over old ones and spent more time in modified sections of the source code, with the goal of maximizing a particular coverage criteria.

We envision a similar integration of our proposed technique into the software development process in the future, with every new version committed to the repository being compared to its predecessors. Using our approach, we can both identify and rank the user's modifications and as Fraser et al. [19] have concluded, we must also be able to communicate what we test and why we test it, in addition to achieving high code coverage. Having understood the effects of the programmer's changes, we can guide a test generator towards sections that behaved differently. In a continuous integration context where every new version is being tested and integrated into an actual execution environment, we would imagine that such test cases could be presented to the developers first as sort of a reviewing mechanism. The generated testsuite would emphasize the effects of the commit, providing a different perspective on what has been implemented and changed. Since a typical commit consists only of a limited amount of new and updated classes and we generate tests focusing on changes, the scope of what the user has to inspect is akin to what he or she has implemented. Once inspected and verified by the user, the generated tests can be added to the project's integration testing suite, hence any future changes that would distort the implemented and now verified behavior would be detected. Employing our approach in a fully automated setting is possible as well, if the goal is to maximize e.g. a selected coverage criterion, we could use the knowledge we obtained on the impact of that version's source code changes to smartly allocate our testing time budget based on our ranking. Unlike Campos et al. [15] who take into account a larger part of the project's history, we use a metric that focuses only on one

single version and its predecessor. In contrast to their approach however, we concentrate on actual code modifications in greater detail.
This thesis is only a first step towards employing such a technique in an actual software development process. Since we focus on obtaining the relevant impact ranking for source code changes, more research is required to conclusively assess the usefulness and consequences of such an integration, in particular a test generation approach leveraging our information about a modification's semantic source code changes must be conceived. As such a specialized test generator is out of scope for this thesis, our main contribution is the implementation of the presented impact analysis in the form of *jHisteg*, discussed in the next chapter.

4 Technical Realization of jHisteg

The last chapter outlined how this thesis proposes to assess the impacts of code modifications made by the programmer, with the long-term goal of automatically generate tests with maximum impact. In this chapter, the thesis' practical contribution is presented, the implementation of the proposed technique as a Java 7 [29] program named *jHisteg* (from **his**tory **te**sting **g**uidance), capable of computing testing targets with maximized impact for other Java 7 projects.

We identified a number of key requirements for our tool, aside from realizing the technique presented in chapter 3 correctly of course. First and foremost, it was important to output results that were generator-agnostic, meaning that instead of employing a particular automatic test generator to output tests, our *jHisteg* tool produces what we call "testing targets". Such a testing target encodes our findings on what to test in a general format instead of feeding it directly to a particular generator. While we used *EvoSuite* [18] in chapter 5, you can in principle use the results of *jHisteg* to guide any test generator. *jHisteg* suggests what or where to focus your testing resources, how these suggestions are used is up to the test generation framework. For example, if one generator only takes method-level targets while another is able to leverage information about which branch should be targeted or if a particular instruction is to be reached, they will probably produce different tests and make use of our results to varying degrees.

As a consequence of this requirement, we realized that *jHisteg* has to output all of its findings in a format that was easily machine-parsable while remaining as human-readable as possible for convenience. We chose the JavaScript Object Notation format, *JSON* for short [31], as it achieves this tightrope walk better than the alternatives in our opinion.

We furthermore decided to persist not only the final output but the results of each and every analysis step to allow for maximum flexibility. This means that data for each intermediate step is available for both confirmability and recombination in the event that one is only interested in a subset of *jHisteg*'s capabilities or want to leverage the results in a different way. For example you could replace our metric for change impact approximation with a custom one with the same notion of syntax and semantic changes.

We have decided to use an abstracted format for all stored information, as this makes the output of each step independent from the internals, e.g. which type of repository was mined. It is our hope that these design decisions will enable further development of our tool and will allow its usage in future work with little additional effort required.

Finally our last requirement is a courteous one, *jHisteg* should be usable on a user's target project without making code modifications or prior familiarization with its internals. This necessitates a documented user interface to configure and control how our tool should run. We have chosen a command line interface to enable *jHisteg* to be used in a scripting environment.

The main arguments that must be provided to *jHisteg* are a software repository and a list of versions from that repository on which to work on. The output for these arguments is a list of suggestions which code sections to test for each of these versions based on what changed between those versions, ordered by the gravity of the changes' impact on overhaul program behavior.
A full execution of *jHisteg* consists of roughly six steps: (1) First, we extract and persist all required versions of the project under test on the hard drive. Once this has been completed, (2) we build the individual versions if necessary, making sure that we have both uncompiled and compiled sources for each version at our disposal. Having now obtained full snapshots of the project at different points of its development time, (3) *jHisteg* makes some modifications to the compiled source code in order to capture information about the version's execution later. This instrumentation of the subject versions makes it possible to generate trace data by executing the project normally due to its compiled code now dynamically outputting that data as it runs. Next, (4) the syntax change analysis is performed, revealing the syntactical changes of one version when compared to its predecessor. Since we require the semantic effects of those syntactical changes, (5) *jHisteg* generates and compares execution traces next, which are obtained e.g. by running a test suite. Note that trace generation can also be performed manually by the user due to the nature of our instrumentation. Finally, (6) we combine the results of our syntax analysis with the trace data to approximate the impacts of the programmers modifications in each version, yielding our testing targets.

The rest of this chapter describes the implementation of *jHisteg* in more detail, illustrating the different extraction and computation steps and explaining how the approach presented in chapter 3 was implemented.

4.1 Version Extraction

Given our repository location and a list of versions, our first task is to extract those versions from the repository and store individual snapshots of the project corresponding to that version on the hard-drive. Our extraction API has been designed with extendability in mind, with the widely used distributed version control system *Git* [21] being fully supported and serving as a full example implementation. Limited support for *Apache Subversion* repositories [5] was also realized, although this is limited to the repository's main branch at time of writing.
The versions to extract must be provided in the form of *identifiers*, which is our general term for the modality of version identification used by the VCS in question, e.g. for Git the user would use a commit's hash. The repository location can either be given as a local path if the repository happens to be available on the user's machine or as an address from which the repository can be accessed. In both cases, the history is not altered and only read access is required.
jHisteg uses the Java Process Framework [30] for interacting with the VCS, hereby directly using its API to achieve the extraction of snapshots. If for example a number

of versions from a repository at "*git@git.myserver.com:myrepo.git*" are to be extracted, then a standard *git clone* operation will be executed through your local Git installation as a first step. This means that authentication and authorization are all handled by the VCS directly and are transparent for our tool as we take advantage of the environment on the user's machine. If a user is currently allowed and able to mine a given repository, he or she can also run *jHisteg* on it as our tool essentially performs the same operations that would be required by a manual checkout.
The actual extraction of the desired version is realized in a similar fashion, our tool directly employs functionality of the VCS to reach a certain point in the development history and then makes a full copy of the current state of the local repository. *jHisteg* uses a directory-based approach for its output, meaning that the chosen output directory is now identified with the given repository and its history, and all future results will be persisted at this location. To access this information in later steps, we extract a list of all versions currently present in the repository, which is stored in the so-called *history*-file (see figure 4.1 for an example).

```
[
   {
      "author": "Elias Hartz",
      "date": "Tue Nov 11 15:31:27 2014 +0100",
      "identifier": "2189a143aeb3884ae451eec1bb6ea71d4391053c",
      "index": 0,
      "msg": "Initial Test Class, initial commit into this repository!"
   },
   {
      "author": "Elias Hartz",
      "date": "Tue Nov 11 15:33:05 2014 +0100",
      "identifier": "07492ff26e3db7648ff71caab8e5f96b3153a16d",
      "index": 1,
      "msg": "Modified the (one and only) Class"
   },
   {
      "author": "Elias Hartz",
      "date": "Tue Nov 11 15:39:44 2014 +0100",
      "identifier": "4c81252a4b93c06db49b45e5b3428714c88a5522",
      "index": 2,
      "msg": "\"Add new Class, modify Access-Level for a field in old!\" he said..."
   },

   (...)
]
```

Figure 4.1: Example of information contained in a history file (Excerpt of 3 versions)

While the concrete contents of that file of course vary from repository to repository and there may be some differences in the formatting of e.g. a date, concepts like the author and the associated message of a particular commit or revision can be found in all major software repository implementations. To have a stronger sense of ordering independent from the identification mechanism of the VCS in question, we introduce an index for

each version, with the initial one being associated with 0. Our extracted snapshots are persisted inside directories with that index as their names. Once the version extraction phase has concluded successfully, the project history along will all versions specified by the user have been persisted into *jHisteg*'s output directory (see figure 4.2).
At this point, we can already compute an early list of changes based on the VCS's output. These *file-level changes* encode little more than the fact that the user modified something inside a particular file, or that a file was added or removed in this version. Still, this information is useful as it identifies candidates for our later analysis steps. We output these file level changes inside each version's directory.

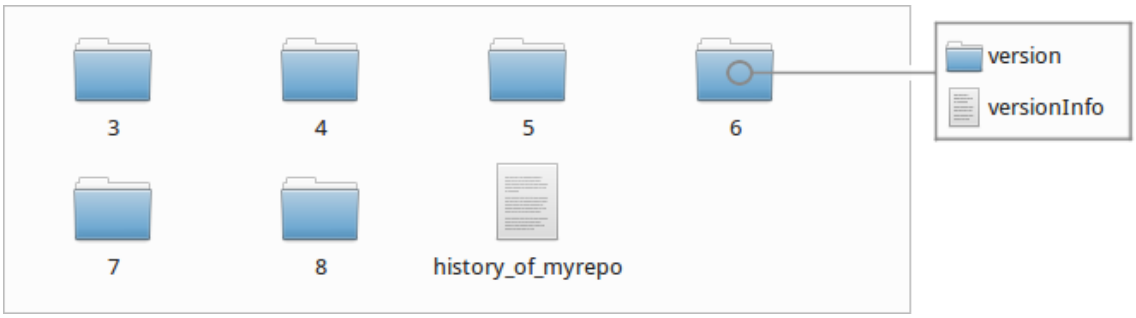

Figure 4.2: Typical content of a *jHisteg* output directory. Each of the numbered directories contains a full snapshot of the version with that index in the history.

Note that the given list of versions chosen by the user further defines the subjects of *jHisteg* analysis as well as their ordering. We deliberately made no assumptions on the sequential arrangement of the provided versions, as such nothing prevents you from providing a reversed list or a completely arbitrary set to analyze. If for example a list of non-subsequent versions is used, our tool will still try to assess what was changed between those versions and approximate the impact of those changes, the difference being that everything that is being detected might be an aggregation of actually distinct development steps. Consequently, when analyzing a particular version, it would be compared against its predecessor in the list given by the user and not its actual predecessor in the history of the repository.
We did not exclude this usage of *jHisteg* as there might be interesting aspects to this behavior, though this is future work. In our experiments, we used only subsequent versions (see chapter 5) which we considered the standard use case for our tool during its development.

When this phase concludes, we have created our directory structure containing independent snapshots of the versions specified by the user, as such we have access to the uncompiled source code.

4.2 Source Code Compilation

In section 3.1 we listed information about the project's general structure among our requirements, in this phase this information comes into play for the first time. For later steps, we must have both uncompiled and compiled sources for each version at our disposal. This necessitates that we know where both are located in the repository and that we know they are adequate, meaning that the compiled source present was actually derived by compiling the current state of this version.
Aiming for compatibility with a wide range of projects, we decided to solve the first problem by requiring the user to state all necessary locations inside the repository. We also included mechanisms to deal with cases in which such locations had changed over the course of development, e.g. compiled code being placed in a different location from certain version onwards. The data defining such locations is stored inside each version's directory, alongside with other parameters for later steps (see figure 4.3).

```
{
   "author": "Elias Hartz",
   "date": "Tue Nov 11 15:33:05 2014 +0100",
   "identifier": "07492ff26e3db7648ff71caab8e5f96b3153a16d",
   "index": 1,
   "msg": "modified the (one and only) Class",
   "comparingAgainstVersion": "2189a143aeb3884ae451eec1bb6ea71d4391053c",
   "sourcesIn": "/home/elias/extractedVersionsFromRepo/1/version/src"
   "compiledSourcesIn": "/home/elias/extractedVersionsFromRepo/1/version/build",
   "commandToBuild": "mvn compile -DmagicFlagOnlyUserKnows=true",
   "buildLocation": "/home/elias/extractedVersionsFromRepo/1/version/poms",
   "exportHumanReadableBytecode": true,
   "renamingOperationsIncluded": false,
   "ignoredClassesByPrefix": [],
   "ignoredClassesBySuffix": [
      "Test",
      "Testsuite"
   ],
   "ignoredMethodsBySignature": [
      "toString()Ljava/lang/String;",
      "hashCode()I",
      "equals(Ljava/lang/Object;)Z"
   ]
}
```

Figure 4.3: Example of an version information file

The second challenge, the adequacy of the compiled code, was circumvented by simply compiling each version anew. This was in large part motivated by the fact that many software repositories simply do not include their compiled code, requiring users to build it themselves. As above, we require the user to provide the required information and if necessary, this can be done on a per version basis. Analogous to version extraction, *jHisteg* is capable of invoking external build systems like *Apache ANT* [3] or *Apache Maven* [4] installed on the user's machine to obtain compiled source code. As Marinescu

et al. [39] noted, we had to make sure that *jHisteg* could handle versions with compilation errors. In addition, we have implemented the option of including the build system's output in *jHisteg*'s log to enable debugging of any issues that might arise.

Once this phase has concluded successfully, we have access to all required resources. However if we were to run a version now, we would not be able to observe its execution as our project under test obviously lacks functionality required by *jHisteg*. Injecting exactly this functionality is the purpose of the next step.

4.3 Source Code Instrumentation

jHisteg is supposed to work on arbitrary Java 7 programs, as such we cannot assume the presence of any functionality to observe our subject version's execution. Since we had very specific intentions on what we wanted to observe (see section 3.2.2 in the previous chapter), we decided to add this functionality into our subject project by making slight modifications to each version's code base while keeping the code's original semantics intact. This in-place instrumentation has the benefit of being transparent to the user, which means he or she can invoke any parts of the program as usual and no special requirements must be considered. If a testsuite is to be started, you can proceed as normal and the program will behave as normal, with the exception of outputting a sort of "transcript" at the end that details what was executed exactly.

Knowing the location of each version's compiled code already, we can add a number of compiled classes from *jHisteg* and modify the original code to make use of that new functionality. This is achieved by using the *ObjectWeb ASM Framework* [7] which provides a small and efficient API to modify bytecode, the instruction set of the Java Virtual Machine. As we will instrument the subject version in-place, thereby making modifications to what was extracted before. *jHisteg* automatically creates a backup of all compiled sources before attempting the instrumentation.
Our overall goal is to observe which methods are called and how they behave internally, as such our modifications roughly fall in two categories, namely storing method entries as well as exits and storing all instructions that are executed: To achieve the first goal, we have to insert code of our own prior to the first instruction of each method's body, as well as just in front of every return-statement contained within. In both cases that code is a method call to *jHisteg* functionality injected to the version's code base, which stores the called method and its parameters in the first case and registers what the method has returned in the second. Intercepting the returned object inside the method is always possible since we perform the instrumentation on compiled sources: In bytecode, all return-instructions pop the value to be returned from the top of the stack, that means just prior to the return, that value is always on top of the stack. Injecting our code there hence works for all cases. In both cases, we furthermore decided to record which thread invoked our injected observation functionality and are consequently capable of distinguishing between multiple parallel code executions.

Note that we must be careful on how we record objects that are returned or passed as parameter. Remember that we are unable to make any assumptions on which kind of objects might be encountered, thus we can base our detection of differences in objects on characteristics shared by all objects alone. For Java, this means we need to rely on functionality provided by the super-class `Object`. Standard methods like `toString()`, whose "*result should be a concise but informative representation that is easy for a person to read*" [28] are intended to identify and communicate the state of an instance. We face a challenge here because these identifying characteristics such as the hash code or the string-representation of our objects are by default execution-dependent since `Object` makes use of the instance's internal address in the JVM, which is different in every execution. Considering that the official documentation states that it "*is recommended that all subclasses override this method*" [28], we decided to only record object characteristics returned by overwritten versions. Inspired by the work of Schuler and Zeller [51], we choose to store the fully qualified class name of an object, as well as its string representation and hash code, provided they are not obtained through `Object`'s default implementation.
Capturing the behavior of a method was defined in chapter 3 as storing which instructions are executed and how often. To this end, we have to insert code for every instruction that records that this instruction was just visited and thus executed. As a method body simply consists of a finite list of bytecode instructions, it is sufficient to store the index of the executed instruction to accomplish our goal. To minimize the need for a source lookup when the trace is generated though, the instrumentation performed by *jHisteg* will also store the type of the executed instruction. Note that concrete values will not be persisted, e.g. we remember that the program executed an `LDC` instruction to load a constant value but not which value was loaded. Akin to above, the thread is also recorded to support multi-threaded applications.
Furthermore, this phase also outputs the call graph of the source as well as a full human-readable version of all instrumented bytecode in JSON for reference in subsequent phases, though the later export can be disabled if your code base is too large or there are security concerns. Both of these datasets are intended to enrich our later computation by increasing its context-awareness. If the subject project was compiled with debugging flags for instance, we can derive a mapping between the lines of source code in Java and the instrumented bytecode, enabling more informative and user-friendly output.
Finally, we need to make sure that the observed execution data is being persisted, otherwise it would not be accessible in later steps. This has been implemented by using a runtime shutdown hook, as a consequence the trace data will be exported into the version's directory just before the JVM terminates. Thus, any execution of an instrumented version will now result in trace data being stored on the hard-drive, be it through *jHisteg* or manually initiated by the user through other means.
It must be stressed that this phase only instruments each of the versions, no traces are being generated yet.

To allow for better control on where exactly the execution trace data is generated, *jHisteg* offers a number of commands to ignore certain classes or specific methods. For

example if a testsuite is used to get traces, it is likely that you are interested only in those sections of the trace that reside inside your actual code base and are not part of the test. Furthermore it is highly recommended to ignore a number of methods derived from `java.lang.Object`, such as the `hashCode()`-method. If they are overwritten by some class in our project under test, the trace will likely contain numerous calls to such methods that are of no interest or benefit to us as they originate from non-project and as such non-instrumented sections of the code. E.g. `hashCode()` is used by various data structures and if instances of the overwriting class are put into such structures, `hashCode()`-calls will bloat the trace while we gain no additional information.
Because information about what has not been instrumented is important for reproducibility, we store the exclusion settings that were used alongside all other information about a version's structure (see figure 4.3).

4.4 Syntax Change Analysis

Git and SVN are environment-independent version control systems, that means, as Fluri et al. put it, that they "track changes of source code files on a text basis without storing detailed information. In particular, the granularity, the type, and the significance level of changes between two versions of a source code entity are not tracked at all" [17]. While our file-level changes computed during the extraction phase provide us with a useful list of candidates, we cannot extract the syntactic change information we need directly from the repository. As discussed in chapter 3, we require an AST-based approach to sustain language-specific information when comparing two classes.
Fluri et al. [17] propose a tree differencing algorithm for this task and have provided the *ChangeDistiller*-tool, capable of identifying source code changes on the fine-grained level required for our endeavor. Given two pieces of Java source code, *ChangeDistiller* tries to transform the AST of one into the other, using a minimal amount of operations, and derives syntactic differences from that procedure. These reported differences correspond to the modifications performed by the user, the change distilling tool could e.g. report that a parameter was added to method `hello()V` (now `hello(Ljava/lang/String;)V`) in class `mainPackage.HelloWorld` and that parameter is the `String s`. This is exactly the information we need in this context.
Our file-level changes and our knowledge about the subject project's structure comes in handy now, we run *ChangeDistiller* on all uncompiled class files that were flagged as modified and are actually part of the source code. The reported results are processed and adapted by *jHisteg* for our use, converting them into *SyntaxChange* objects. This adaptation encapsulates *ChangeDistiller* and ensures that upon replacing it with a different means of syntax change detection, the rest of *jHisteg* would only be minimally affected.
Note that we perform the syntax analysis on uncompiled code while the instrumentation performed in the previous phase affected only the compiled sources. As such, both operations do not interfere and both phases could be executed in any order. As usual, we also persist our findings into the version's directory in a JSON-based format. Figure

4.4 contains a representative excerpt of such a *syntaxChange*-file obtained by running *jHisteg* on a real-life software project.

```
[
(...),
 {
  "classInPreviousVersion": "org.joda.time.format.DateTimeFormat",
  "codeRemoved": "Map<String, DateTimeFormatter> PATTERN_CACHE",
  "codeType": "FIELD",
  "level": "CLASS_LEVEL",
  "type": "CODE_REMOVAL"
 },
 {
  "class": "org.joda.time.format.DateTimeFormat",
  "codeAdded": "ConcurrentHashMap<String, DateTimeFormatter> PATTERN_CACHE",
  "codeType": "FIELD",
  "level": "CLASS_LEVEL",
  "type": "CODE_ADDITION"
 },
 (...)
 {
  "affectedMethod":
      "getFormatter(Ljava/util/Locale;)Lorg/joda/time/format/DateTimeFormatter;",
  "class": "org.joda.time.format.DateTimeFormat$StyleFormatter",
  "codeBefore": "f = DateTimeFormat.forPattern(pattern);",
  "codeNow": "f = DateTimeFormat.forPattern(getPattern(locale));",
  "codeType": "ASSIGNMENT",
  "level": "METHOD_LEVEL",
  "type": "CODE_MODIFICATION"
 },
(...)
}
```

Figure 4.4: Excerpt of detected syntax changes for two subsequent versions from the JodaTime [32] project

Using *ChangeDistiller* enables us to obtain syntax changes for modified classes, so how do we deal with new or removed classes? The later, the deletion of an entire class, is straightforward: Since that class has been removed, there is no need to test it anymore. If functionality provided by that deleted class was used elsewhere before, the programmer must have modified those sections because otherwise the code would not compile and we would have aborted our analysis for this version. Those modifications are of course detected and analyzed, as they cause their containing class to be flagged as modified. In the event that no such changes exist, the deleted class contained only code that could not be reached from the rest of the program.
Dealing with newly added classes requires a different approach, as their new functionality is of great interest from a testing perspective. While its usage in existing code can be analyzed with *ChangeDistiller*, obtaining detailed syntactic information with this tool requires two ASTs to be provided, of which we have only one. Fortunately we can treat

any functionality contained within this class as a code addition, that means we do not require a detailed analysis as long as we remember that this class has just been added. Should we encounter an execution trace using this class in a later phase, we will need to treat the relevant part of the trace as being inside syntactically modified code. To that end, we add a *SyntaxChange* object on class-level affecting the entire class to our existing list of detected syntactic modifications.

While being a potent tool, *ChangeDistiller* has some shortcomings one must be aware of. As there are potentially multiple ways of transforming one AST into another, there may be mismatches in one area of the tree that might propagate to others, affecting the choice of transformation operations. In addition, the distilled changes might not match the programmer's mental model, especially when it comes to source code replacement. The data presented in figure 4.4 for instance contains an addition and a removal of a class's private field. Considering the fact that this particular version's message was '*Using ConcurrentHashMaps rather than synchronized Maps.*', the programmer would likely expect a single code modification to be reported. Finally, *ChangeDistiller* will not detect if the value of a constant has been changed as long as the statement remains intact, since it does not take the concrete value of its AST nodes under consideration (see figure 4.5).

```
1 public class Main {
2
3   public final String str = "Lalala";
4   public final int num = 12;
5
6   public static void main(String[] s){
7     (...)
8   }
9 }
```

```
1 public class Main {
2
3   public final String str = "Lalalala";
4   public final int num = 4+str.length();
5
6   public static void main(String[] s){
7     (...)
8   }
9 }
```

Figure 4.5: *ChangeDistiller* reports syntactic modification for the assignment of field `num` but not for field `str` since the AST of that assignment differs only in the concrete constant value.

By adapting the *ChangeDistiller* tool to be used as a part of *jHisteg*, we have obtained the syntactic changes between all specified versions when this phase concludes. It is worth to note that thus far, all phases have only used static information and analysis techniques. With this in mind, *jHisteg* has been designed to import all previously computed data up to this point on subsequent runs, thereby saving execution time. Our reasoning behind this was to enable the user to experiment with the various options he or she has when generating traces with ease, which we considered an important use case. When running multiple times on the same repository with different configurations, *jHisteg* automatically checks the output directory to make sure that already present information is compatible.

4.5 Trace Divergence Analysis

We distinguished between syntactic modifications and their effects on the source code semantics in chapter 3, with the later informing our analysis of the former's impact. Having obtained the static syntax change data through *ChangeDistiller*, we now require actual execution data from our subject versions.

4.5.1 Execution Trace Generation

Recall that we have instrumented the compiled source code of each version in a previous phase, injecting functionality that causes the program to output abstracted data about its execution. We mentioned before that our observation mechanism is a transparent one, with the idea of making the trace generation required now more intuitive for the user. In consequence, the only thing required to generate the traces we need for our analysis is to start the program. While this can be performed manually of course, you can also supply *jHisteg* with main methods to start, JUnit [34] testsuites to run or commands to execute (e.g. *mvn test* if the project uses *Maven* [4]) to ease this procedure. No matter if being performed by the user outside of *jHisteg* or by our tool, each execution of the subject program will produce a new *observedTrace*-file inside the version's directory. This means you are not limited in the amount of traces or face any restrictions on how you obtain them.
Our *observedTrace*-files contain abstracted execution data, detailing which methods were invoked with what objects, which bytecode instructions they executed and what they returned. Bear in mind that we do not record concrete values when the program is running, only selected criteria of parameters and returned objects are stored.
Figure 4.6 shows an exemplary trace obtained by running a JUnit test belonging to a subject program, with the test being instrumented as well in this case. For better understanding, also consult figure 4.7 which presents the source from which this was derived. Our example *observedTrace* data consists of two distinct traces. We can see that the first trace is only made up of one single observed call to the test class' constructor. While the entire bytecode of the test was instrumented, the constructor originates inside the JUnit library that is preparing to run the test case. When the JVM reaches `tests.SomeTest.<init>()V`, our injected functionality records this and since it is the first observed data, we consider it as an entry point of a new trace. Once the constructor returns, that trace comes to an end since there was no caller recorded.

```
[
  {
    "called": "tests.SomeTest.<init>()V",
    "type": "CONSTRUCTOR",
    "calledWithParameters": [],
    "trace": [
      "[0]> ALOAD",
      "[1]> INVOKESPECIAL",
      "[2]> RETURN"
    ],
    "returned": {
      "class": "---",
      "stringRepresentation": "void"
      }
  },{
    "called": "tests.SomeTest.unitTest()V",
    "type": "METHOD",
    "calledWithParameters": [],
    "trace": [
      "[0]> SIPUSH",
      "[1]> INVOKESTATIC",
      {
        "called": "mainPackage.SomeClass.echoInt(I)I",
        "type": "METHOD",
        "calledWithParameters": [{
          "class": "int",
          "value": "2"
        }],
        "trace": [
          "[0]> ILOAD",
          "[1]> IRETURN"
        ],
        "returned": {
          "class": "int",
          "value": "2"
        }
      },
      "[2]> SIPUSH",
      "[3]> IF_ICMPNE",
      "[4]> ICONST_1",
      "[5]> GOTO",
      "[7]> INVOKESTATIC",
      "[8]> RETURN"
    ],
    "returned": {
      "class": "---",
      "stringRepresentation": "void"
    }
  }
]
```


Figure 4.6: Example of a *observedTrace*-file

The second trace contained within that file starts when the JUnit Framework actually begins testing, the method `unitTest()` is being invoked. As its first real action, this method calls functionality inside our class-under-test, it invokes `echoAnInt` with the parameter 2. Its rather simple purpose is to return the provided integer completely unaltered, which we nonetheless record. Testing method `unitTest()` subsequently compares this returned value with an expected one using JUnit's `assertTrue` statement (see 4.7). Aside from a recorded `[7]>INVOKESTATIC`, this is not visible in our trace since the JUnit library was not instrumented, our trace continues without registering information about that method invocation. Finally, `unitTest()` returns and we record that it returned with `void`.
As this example illustrates, what we instrument has tremendous repercussions on how our traces will look, which motivated our decision to give users of *jHisteg* fine-grained control over which sections of the code to instrument. For instance, the first trace in this example represents a default constructor implicitly present in the class, which is of little use to us in this case. By not instrumenting `tests.SomeTest.<init>()V`, our data would only consist of one trace. In general, it is completely up to the user to choose which traces to generate and how. The recommended way however is to use an unmodified but instrumented testsuite present in both versions, as this will lead to traces with entry points residing in syntactically identical code.

```java
package mainPackage;

public class SomeClass {

   public final int someInteger;

   public SomeClass(int number){
      someInteger = number;
   }

   public SomeClass getObject() {
      return this;
   }

   public static int echoInt(int i) {
      return i;
   }
}
```

```java
package tests;

import mainPackage.SomeClass;
import org.junit.Test;

import static org.junit.Assert.assertTrue;

/* Both this class, as well as
 * the class-under-test have
 * been instrumented. */
public class SomeMinimalTest {

   @Test
   public void unitTest() {
      assertTrue(SomeClass.echoInt(2)==2);
   }

}
```

Figure 4.7: Java Source Code of example in figure 4.6

4.5.2 Trace Matching and Comparison

Now that we have obtained a number of traces for both versions, our main task is to compare this execution data to identify sections in which one trace diverges from another. These trace divergences correspond to the changes in the program's semantic caused by

the user's modifications and, as discussed before, do not necessarily occur at the point of that modification. Our task is twofold: First, we need to decide if it is actually sensible to match a particular trace onto another and then we need to compare the two.
A typical commit includes a limited amount of additions and modifications, it is unlikely that the shape and purpose of the entire program is drastically altered between two versions, that means we should consider two completely distinct traces incomparable. Our automatic approach matches traces based solely on having the same entry point, a design decision influenced by our standard use case as well as our recommended way of generating traces with *jHisteg*. A manual matching can still be provided by the user, should that be necessary.

How do we compare two traces and obtain a metric on how much they have diverged overall as well as exact information where they diverged from each other? As discussed in section 4.3, a bytecode method's body has a finite number of instructions, hence storing the index of each executed instruction would be sufficient to comprehend what was executed since the concrete values of the instructions have been abstracted. We can therefore interpret each method invocation inside a trace as an array of indices into the bytecode source. If we were to use characters to identify instructions, assigning a unique character to each of the method's instructions, we would be dealing with a single string describing the program's behavior during that particular method invocation.
With this in mind, we choose to use the string similarity measure proposed by Levenshtein [38] as our metric. Our implementation tries to transform one behavior into the other by adding, removing or replacing indices, the minimal number of required operations is the edit distance between the two arrays. Given two sections of a trace, we can hereby derive an integer-based metric on how much these sections differ, performing this computation for all method invocations of the trace results in the *trace distance*.
In principle, our approach here is connatural to the core idea of Fluri et al. [17] as both are based on the idea of transformation. However, while they have concluded that the Levenshtein algorithm's property of being affected by ordering changes renders it inadequate for their purposes, we can use it for our analysis as the composition of our index array is dictated by the method's control-flow. Both executions will always start with the first instruction and might start to differ depending on whether branching instructions are executed or not. When applied on syntactically unmodified code, the Levenshtein distance states exactly how much two traces have diverged in number of different instructions, a different conditional branch or an additional iteration of a loop will be perfectly visible.
We obtain the Levenshtein distance through a dynamic programming approach and then extract from the computed matrix both the numeric distance in instructions, as well as the number and exact locations of divergent sections. Furthermore, we keep track of any differences in the observed characteristics of all parameters and returned values. To recurse into nested method invocations, we match the invocations of both traces in a separate step, using again our transformation approach. This ensures that if e.g. a method invoked two other methods in the first trace while invoking one additional one in the other, we can still compare data present in both traces. Any additional or missing

method call is noted as well of course.
When the trace comparison phase concludes, we have computed both numerical metrics as well as precise information on how much our traces differ in number of instructions, on how often they diverge from another, and on the similarity of objects passed around and methods invoked. This information is persisted in the form of *TraceDivergence* objects.

4.6 Change Impact Approximation

In this final phase, we combine our syntactic change data with the numerical metrics obtained previously in order to output testing targets, suggestions on what to test and in what order to test it based on the semantic impact of the change.

As discussed in chapter 3, the general idea of our impact analysis is to map all observed semantic differences to the responsible syntax modifications in order to approximate the size of impact of these modifications. In the current version of *jHisteg*, this mapping is method-based: As a first step, we aggregate all syntactic changes affecting the same method or class. Consequently, this results in exactly one list of class-level changes for each class and individual lists of method-level changes for each of the class' methods. As our second step, we assign our trace divergences. Each divergence corresponds by construction to the behavior of a particular method invocation but, as illustrated in chapter 3, that method may very well be syntactically unchanged.
Therefore, our mapping approach must deal with two cases: If the method in which we observed a trace divergence had syntax changes in this version, we map the divergence onto this method as it was likely caused by its altered behavior, otherwise we check which syntactically modified methods invoked the divergent but unmodified one. Should such syntactically altered methods exist higher in the invoked method's call chain, the divergence is associated with all of them as we have no means to determine if a particular syntactic change had no effect. If the call chain on the other hand contains no methods with syntax changes, the divergence remains unmapped. Such observed divergences without syntax changes are usually caused by a different object state for which in turn other mapped divergences are responsible, we have to record them separately.
In short, our results have now been restructured into (a) a set of syntactically unmodified methods in which we observed at least one divergence and (b) a set of methods with syntax changes to which we have associated divergences localized inside the method itself along with those which were observed in unmodified methods invoked by the modified one. By construction, the sets are distinct in terms of methods.
From these sets, we create *TestingTarget* objects for each of these methods next. These will later contain a single numeric value, the higher this value, the stronger is that modification's impact and as such the higher is the recommendation to retest this method. Given a method from either the (a) or (b) set, a *TestingTarget* object is defined by three lists: The list of the method's syntax changes, the list of observed divergences of this method and the list of divergences detected in unmodified methods called by this one. We denote the two later lists as local and non-local impact. One must be aware of the

fact that these lists contain data acquired through dynamic means, they are holding divergences from all analyzed traces and hence depend greatly on how the user chose to generate them before. If the testing target's method comes from the (a) set, both the list of syntax changes and the non-local impact list will be empty. If it is contained within the (b) set, we will always have at least one syntactic change but both the local and non-local impact lists may be empty independently from each other.

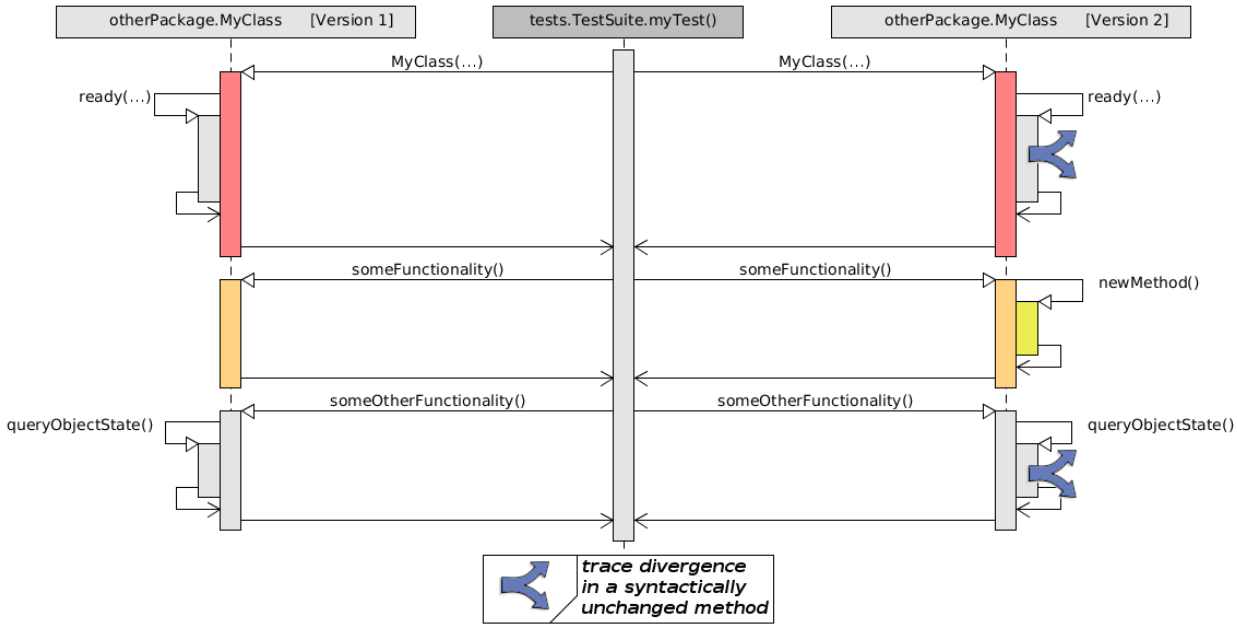

Figure 4.8: A testsuite creating an instance and calling functionality

To elaborate on how *TestingTarget* objects are derived, we will now discuss a concrete example given by figure 4.8, which contains a sequence diagram of a typical test case execution. The test, represented by the method `myTest` inside the class `TestSuite`, creates an instance of `MyClass` using its constructor and then invokes functionality of that instance. The test's assertions and internals have been omitted, instead figure 4.8 shows the class under test twice, representing the tests execution in two of our subject versions. Colored methods, e.g. the class' constructor, have been modified from version 1 to 2. Recall that we possess the syntactic change information at this point and that both of these executions have been observed and are accessible in the form of traces. Generating *TestingTarget* objects means combining these two datasets, we will discuss this process along the test's execution for exemplary purposes.
In both versions, our trace starts in `myTest` which is present in both versions and was not modified. Our two traces can be matched by the automated matcher since they share the same entry point. The test case's first action is to call the constructor of `MyClass`. This constructor was modified, as such we will create a testing target for it. All divergences originating inside this constructor make up the target's local impact, these are caused by the user's modifications directly. For instance, fields may be initialized in a different

manner. At some point of its execution, the constructor further invokes the method `ready` in both versions, which has not been changed by the programmer. Any trace divergences observed in `ready` are therefore mapped to the constructor, which is the only syntactically modified method higher up in the call chain. For this reason, they make up the non-local impact of the constructor's testing target. These divergences could have been caused by, for example, different parameters being passed to `ready` or the above mentioned modifications of the way the `myTest` instance is initialized.
The test's second action is to invoke functionality of the newly created instance. Akin to above, we will create a testing target for the method `someFunctionality` since it has been modified by the programmer. What makes this method interesting is that it calls another method, one that was not called in the previous implementation. The target's local impact includes this information, along with all other trace divergences that occurred inside `someFunctionality`. Since we have no section in the trace of version 1 we can compare the execution of `newMethod` to, no divergences can be found which means that the target's non-local impact list is empty.
Finally, the test case calls `someOtherFunctionality`, which in turn calls the method `queryObjectState`. In our example, the two traces diverge during its execution. As the method with the divergence is syntactically unmodified, we examine the call chain. In contrast to the `ready` method however, no methods with syntax changes are present since both `someOtherFunctionality` and `myTest` were left alone by the user. We therefore create a testing target for `queryObjectState` that has no syntax changes and only local-impact. While the method's name implies that the observed divergence could have been induced by the previously detected differences in how the constructor of `MyClass` operates, this is not certain. Including these divergences in the non-local impact of the constructor could be false, while our strategy of generating a testing target for `queryObjectState` ensures its differing behavior is registered on its own merits.
To summarize, we would end up with three *TestingTarget* objects for the example given by figure 4.8, with two pointing to syntactically changed sections while one suggests that the unmodified method `queryObjectState` is to be examined again. What is left to do is to order these three testing targets by their significance.

Given two versions, we can now decide which methods we want to test as we have identified what the user changed and what was affected. Having derived an unordered set of *TestingTarget* objects by combining syntax and semantic changes, our final task is to provide a ranking of these methods according to their impact. As such, *jHisteg*'s final output will be a sorted list of these testing targets.
To achieve our ranking, we can combine our previous metrics such as e.g. the amount of divergent sections or the trace distances which were computed for an individual trace as described in section 4.5.2, to form a single metric value for an entire testing target. As the *TestingTarget* object holds data from multiple traces and the method that we are analyzing is by construction part of all of these traces, we first need to derive average values for these metrics over all of the traces. Note that we must perform this averaging for both our local and non-local divergences, hence we end up with two sets of average metric values: The first set contains the average number of differences in returned

objects and parameters, the average number of different method invocations as well as the average amount of divergent code sections, and the Levenshtein distance averages between relevant code sections derived from the local impact divergences. The second set holds the same type of information computed from the non-local impact. In addition, we extract the number of detected syntax changes and how many distinct methods were affected overall, with the former being interpreted as another local metric while the later is a non-local one.
Holding meaningful averaged metrics for the *TestingTarget*'s method now, we scale the individual values: Primarily, our chosen scaling emphasizes all non-local values over our local ones because our goal is to rank the user's modifications based on their impact on the entirety of the program. Furthermore we stress the amount of divergent sections, while downgrading the actual trace distance to little more than a tie-breaker. The reasoning behind this is that the number of divergent sections captures control-flow differences better than distance (or coverage for that matter), ignoring the actual length of the alternative branch. All non-local values are further accentuated in a second step as we take the call distance between the modified method and the divergent method into account: Akin to Schuler and Zeller [51], we use the minimal amount of calls necessary to reach the method with the divergence in lieu of the trace's call chain as this provides a usage-independent understanding of how strongly connected these two code sections are. Recall that we are in possession of the entire call graph of the subject version from the instrumentation phase (see section 4.3), hence this value can be computed in the current phase.
Note that if a syntactically modified method has no local impact at all, this might indicate that the programmer's modification had no effect or was not executed. Since our abstraction does not include the concrete values of a method's execution, that is not a reliable detection mechanism. Important for us is that compared to another method with similar impact on the rest of the program, the method's ranking will be lower as no local metric values contribute.
Finally, we are able to determine the overall ranking value by adding all of *TestingTarget*'s scaled local and non-local averages together. Sorting the list of all of these objects for the given version based on their rank yields *jHisteg*'s expected output: We have obtained an ordered list of testing targets, suggesting what to test and in what order to test it based on the semantic impact of the user's changes in this version. This result is persisted in form of the *testingTarget*-file inside the subject version's directory. Figure 4.9 contains an annotated screen-shot showing a version's directory after our tool has terminated.

While we emphasize the amount of divergent sections in our default metric, *jHisteg* also supports two alternative approaches which shift the focus on the actual number of different instructions executed: The first alternative is a coverage-based metric inspired by Schuler and Zeller's *Javalanche* [51]. In this mode, *jHisteg* replaces trace distance with differences in coverage, we record for each trace how often each instruction was executed and then compute the differences on a single instruction basis. For a given method invocation, its coverage metric value is equal to the sum of all of its single instruction

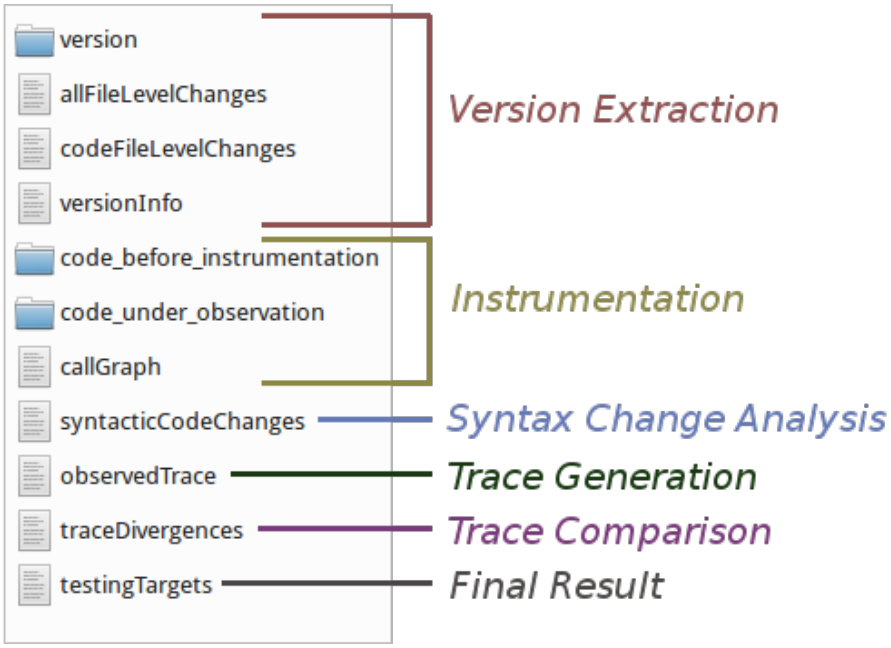

Figure 4.9: Typical content of a version directory after *jHisteg* terminates.

differences. Note that this data can also be derived from our trace format, we do not require a different instrumentation to obtain these values. The second alternative is to only compute the trace distance, skipping the computation of the divergent sections. This stresses the amount of different instructions, basically using only the pure Levenshtein distance to rate how different one trace is from another. In contrast to coverage, the ordering of executed instructions is still taken into account. Both alternative metrics can also be combined, in which case their values are simply added together.
Manual investigation of testing targets generated using these alternative metrics have resulted in rankings that did not satisfy our overall goal. However since no automatic test generator with the capability to fully leverage *jHisteg*'s functionality exists at time of writing, we have to consider the research question on choice of suitable metrics unanswered anyway and have thus chosen to include our previously implemented metrics in *jHisteg* to increase its flexibility and potential for future work.

5 Evaluation and Discussion

We have introduced both the high-level concepts behind *jHisteg* as well as its concrete implementation in the previous chapter. Now, the actual capabilities of our tool must be evaluated. Here, we face a challenge since our long-term goal of generating better tests cannot be reached within the scope of this thesis. While we would like to examine if the practical issues pointed out by Fraser et al. [19] can be mitigated by generating tests based on the history of the subject project, we do not possess an automated test generator capable of leveraging our data.
While we are able to run existing generators, for example *EvoSuite*, on e.g. all method-level testing targets reported by *jHisteg*, such generators do not leverage our ranking or information about divergences and concrete syntactical changes. This means, *EvoSuite* basically behaves as normal and the findings of Fraser et al. [19] still apply to these generated tests.
We nonetheless implemented an adapter for *EvoSuite* in order to verify that *jHisteg*'s results can be used to guide automated test generation tools in general. Without the capability of incorporating information that goes beyond testing only specific methods however, there is currently not much to be gained from this adapter. One could attempt to alter the allocation of limited testing resources based on the ranking akin to continuous test generation as proposed by Campos et al. [15] but this, as mentioned above, would not enable *EvoSuite* to make proper use of the computed data. Furthermore, the impact of a modification may only be visible in a certain execution trace, e.g. if a specific object state must be achieved, and there is no guarantee that *EvoSuite* would be able to reach the exact state when the syntactic modification makes its potentially defective impact.

With this in mind, we decided to focus our evaluation on the concrete results produced by *jHisteg* at the moment, aiming to answer the following research questions:

- Are the design decisions behind *jHisteg*'s architecture compatible with what is present in real-life software projects?
- Do we observe a version's execution at the right granularity?
- Does our tool produce a meaningful ranking that is in fact representative of the impact of the programmer's modifications?

5.1 Setup

Our first step was to select a number of subject projects to work on, as well as what versions to include. To avoid bias, we followed the strategy by Campos et al. of selecting popular open source projects. In fact, we choose to put *jHisteg* to work on a subset of the subjects used in their study [22], selecting the first three in alphabetical order as these were thematically distinct. At this scope, the planned manual examination was still reasonable, furthermore their remaining subjects were thematically related to the three we had already selected:
Our first subject is *http-request* [27], a library containing only a single class that provides an API to make HTTP requests and access the response. In contrast to this adapter-like project, our second subject *JodaTime* [32] is more of a framework, as this library is intended to provide a "quality replacement for the Java date and time classes". For our third subject, we originally intended the *JSON in Java* [13] project to represent a data-structure oriented program in our evaluation but it turned out that their repository was inadequate for our use case: While we could extract and compile versions with *jHisteg*, there were no means of generating execution traces as both tests and main methods were absent for the majority of the development history. In the end, we used *jsoup* [33] instead, an HTML parser providing functionality that serves a similar purpose, namely extracting, selecting and modifying elements inside a complex, hierarchical data structure.
When deciding which versions to use, we again followed the example set by Campos et al. and chose the same commits as they did for their evaluation, which allowed us to start without any prior examination of our subjects. We used *jHisteg* to extract and analyze 100 subsequent versions of *http-request*, *JodaTime* and *jsoup* each and examined the resulting testing targets, comparing them with the actual modifications made in that version. Furthermore we were interested in the overall amount of targets generated and how many of those targets were derived from trace divergences only.

5.2 Results

Overall, *jHisteg* ran on all three projects successfully and was able to generate data on both syntactic and semantic changes, the later by using the subject project's testsuite. All projects contained versions with actual source code modifications as well as versions where no such modifications took place, in which e.g. only a document present in the repository was modified. *jHisteg*'s memory requirements and runtime were moderate for *http-request* and *JodaTime*, with the compilation phase being the most time consuming. In the end, their output directories, which contain all 100 snapshots along with our generated data, were roughly 1 and 10 gigabytes in size respectively.
For *jsoup* on the other hand, the execution was more demanding: While all compiled and instrumented snapshots of *jsoup* make up about 6 gigabyte in total, the generated execution traces raised that number to over 100. The traces also had to be generated individually on a test-by-test basis to minimize the memory demand and we had to

exclude two test cases as they overburdened our testing machine which featured 7 gigabytes of RAM available to the JVM. With file-IO being a limiting factor in this case, *jHisteg* ran significantly longer but still successfully computed testing targets. Considering our experiences, we can safely say that *jHisteg* is in fact compatible with real-life software projects, thereby answering our first research question. With all experimental data available, we proceeded to inspecting our results in detail.

Figures 5.1, 5.2 and 5.3 show an overall comparison between the number of methods that are to be tested for each version, derived either naively by testing every class that was flagged as added or modified by the VCS, or derived from *jHisteg*'s testing targets. Naturally, the amount of syntactically modified methods is equal or less than the number obtained by the naive approach, the sum of all reported testing targets however can be larger due to the fact that methods with divergences that reside in unmodified code might be included. As expected though, we have notably less to test on most occasions when compared to the naive approach which is in itself an improvement.

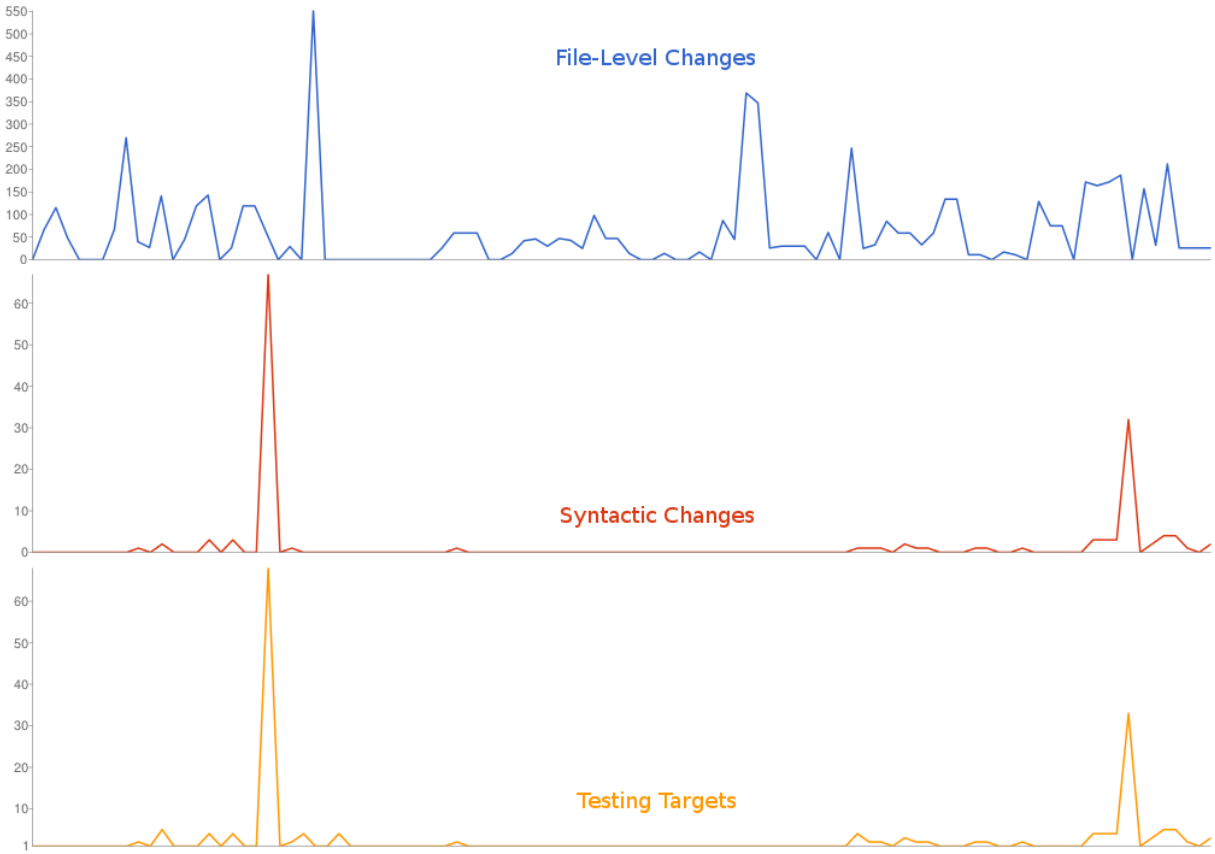

Figure 5.1: Number of methods to test for *JodaTime*

What catches the eye are the differences between the detected syntactic changes and reported testing targets, for all three projects we consistently have more targets than syntax changes. As discussed in chapter 4, these unmatched divergences may result

from modified objects states or other concrete value changes that are not captured in our abstraction.
Examining e.g. figure 5.2 more closely however shows that we also report targets for versions that had no file-level changes whatsoever, on occasion quite a high number to be precise. Our manual inspection of targets and the observed traces revealed that we detect divergences in parameters and returned values inside unmodified and in fact unaffected code for almost every version. The *jsoup* results, shown in 5.3, are particularly bad in this respect, *jHisteg* consistently reports over 110 methods that are to be tested. Considering that one has to test only 61.02 methods on average when using the naive approach based on file-level changes, this is discouraging at first. Upon closer inspection, we noted that many of these unmatched divergences result from the environment, for instance we constantly detect parameter divergences in the method `HttpRequest.get` in *http-request* as the port number of the network connection used by the request is a different one each time. The effect of parameter and returned value divergences on the overall metric value of a target is a weak one due to our chosen scaling values that emphasize divergent code sections, however for methods central to the subject's functionality, these divergences occur in large enough quantities to blur our result and subsequent ranking.
Not all of these false positives are caused by environmental conditions, some also originate inside code that simply behaves differently each time: We frequently report e.g. the method `select` in the class `ConverterSet` from the *JodaTime* project as it internally manages an immutable hash-table, thus behaving differently for every execution. In short, due to these divergences our testing targets suffer from a constant "noise" of sorts. If e.g. environmental conditions change, we can observe significant spikes in the amount of testing targets without any user modifications taking place, with *http-request* particularly affected by such spikes (see figure 5.2).
The number of these falsely reported targets of course depends on the project's structure, *jsoup* for example passes all of its tokens through a number of processing methods. Those tokens are detected as divergent since their observed characteristics differ, this is due to their `toString()` implementation including the object's internal address. Consequently, we generate a testing target for each of these methods and since each of them is called frequently during *jsoup*'s execution, they achieve a high ranking value.
There is also a significant gap in our results from *JodaTime*, in which source code modifications took place but no targets are being reported. The majority of these changes actually affect fields of classes and their types instead of code inside methods. Since the file is marked as modified by the VCS, we include the methods of this class in the file-level changes count but they are absent from the number of syntactically modified methods, as none was changed in a way detectable by *ChangeDistiller*. With no method invocations behaving differently in our abstraction, which excludes concrete values, *jHisteg* does not yield any testing targets for these versions.
We also came across several occasions where *ChangeDistiller* had failed to report a detectable but deeply nested syntactic change, a limitation stated by Fluri et al. [17]. While we did detect divergences, the lack of a reported syntactic modification caused a low ranking and lead to other divergences being reported as unmatched instead of being

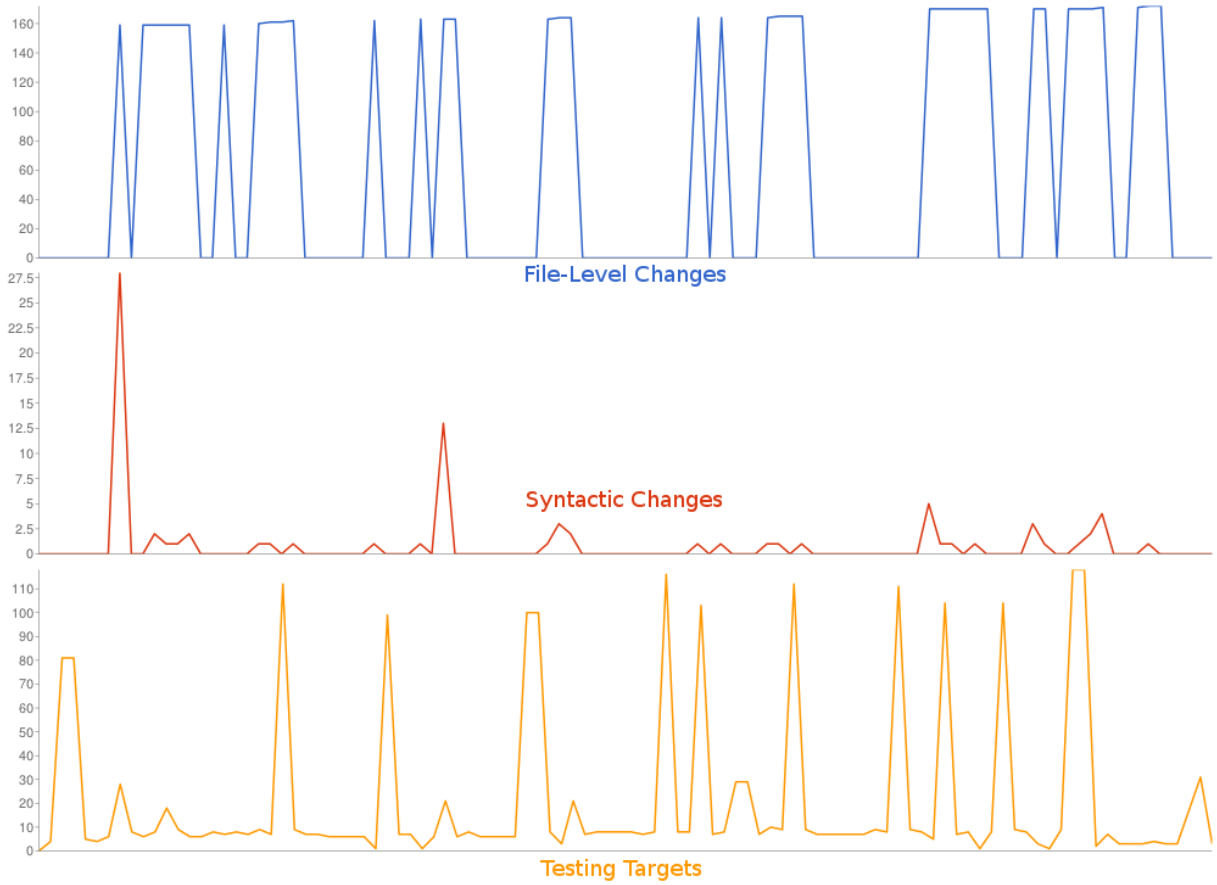

Figure 5.2: Number of methods to test for *http-request*

associated with the modified method.

What is our overall result? Including the noise, one can observe that the number of methods to test is notably lower than the naive approach. This means, we have less to test and can focus our efforts and potentially limited resources more effectively. If one now ignores the testing targets classified as noise, their global amount not only shrinks dramatically but also resembles the number of methods with syntax changes more closely. That means, the majority of observed divergences occur inside modified methods or methods being invoked by them and can as such be matched to a user's modification. Our ranking also resembles our expectations more closely, with the strongly modified methods or the ones calling many other methods rising to the top.
While we can consider our results to be promising overall, one cannot overlook the fact that the current approach of observing which objects are being passed around seems to be inappropriate under certain conditions. While Schuler and Zeller [51] obtained good results when using a similar abstraction to detect equivalent mutants, our evaluation indicates our chosen granularity seems to be oversensitive in the context of assessing a code modification's impact for some projects.

Furthermore, while we were able to select thematically distinct projects without being biased, the same cannot be said about our manual inspection of the generated rankings. Our judgment on this topic is obviously influenced by the fact that we know how this ranking was derived. Considering that, as Fraser et. all put it, "mechanically evaluating automatic test generation tools, without user studies, increases the likelihood of forming misleading or incorrect conclusions"[19], we must conclude that further work is required to decisively answer our remaining research questions.
To recapitulate, *jHisteg* is able to produce ranked testing targets based on the history of real life software projects and these rankings are, provided certain environmental and object-related conditions do not apply, representative of the user's modification and their impact in our opinion.

Figure 5.3: Number of methods to test for *jsoup*

5.3 Threats to validity

The main threat to validity obviously results from the correctness of *jHisteg*'s implementation, as well as its usage during this evaluation. It is possible that unknown defects inside our tool or an operating error could have lead to incorrect conclusions. To

minimize this risk, we inspected both the logging output of *jHisteg* as well as the generated testing targets during the evaluation process and furthermore added an extensive testsuite to our tool.
Our means of deriving such execution traces represent a threat to validity as well, if a code section that is indirectly affected by a user's changes is not executed, we are unable to detect this effect. Using *jHisteg* effectively requires that one is able to actually execute all sections of a subject's code, a requirement that is there by design and should not be underestimated. All three of our subject projects had extensive testsuites and as far as we could tell, they covered the majority of central classes.
Finally, our choice of subjects for this evaluation also does not include any industry-grade repositories. As such projects often face completely different quality standards than open source software, our results may not be applicable in general. Since we did not have access to any suitable Java 7 projects as these typically are closed source, selecting a broader scope of projects was not possible within the scope of this thesis. However, we went to great lengths to select projects and versions without being biased, setting us suitable boundaries to work in, without personally examining the projects in detail. Since our subjects are of different scope and purpose, we do have confidence that our findings are representative of a significant portion of the software projects in the wild.

6 Conclusion and Consequences

Software development is an incremental process in which previous versions of a project are used to construct a new iteration. With intuition telling us that it suffices to test what has been modified or newly introduced during each iteration, we have proposed to focus automatic test generation on sections of the source code that were changed by the user, with a special emphasis placed on the impact of that change on the program overall.
In chapter 3, we have shown how each modification's impact can be approximated by combining statically obtained data about syntactical changes with divergences observed in execution traces. We have introduced the notion of distinguishing between the syntactic and semantic aspect of a change and explained how this information can be combined to both produce and rank testing targets.
Chapter 4 of this thesis focused on our implementation of this technique. We have illustrated how *jHisteg* extracts and compiles subject versions from a repository and subsequently instruments them to enable execution trace generation. The syntax change analysis and trace comparison phases have been detailed, along with an explanation on how our tool derives its testing targets from these datasets.
Finally, chapter 5 showed the potential of our approach by evaluating *jHisteg*'s performance on multiple, non-trivial software projects. This evaluation also pointed out several areas where our technique might be improved upon in the future, such as developing a better way of tracking parameters and return values.

Despite these promising initial results, impact-based test generation guided by a project's history remains in its infancy. The logical next step from this thesis onwards is to develop an automated test generator that is capable of utilizing the data computed by *jHisteg* to the fullest.

Another area one would likely focus on once such a generator exists are the metrics currently employed by *jHisteg*. We rank our testing targets according to metrics designed to capture the significance of each change's impact. These metrics and their associated scaling have been derived and refined from experiments during the development of *jHisteg* but without the ability of procuring runnable testsuites based on that data, no definite verdict on their quality or adequacy can be given. Since we currently lack the means of judging their quality based on user feedback as Fraser et al. [19] suggest, our evaluation focused mainly on the qualitative aspects of *jHisteg*'s current behavior. As a consequence, exactly which types of metrics are most suitable for the long-term goal of history-guided test generation and how well they actually capture the impact of a user's modification requires further investigation by the research community.

Another interesting direction for future research would be to consider the work of Murta et al. [45]: As previously mentioned, the commonly used version control systems of today are environment-independent. If environment-specific systems as proposed by Murta et al. were to become more prominent, this could lead to a point where *jHisteg*'s distinction of file-level changes and syntactic changes would not be necessary anymore. Finally, a subset of *jHisteg*'s capabilities and features could be put to use in other areas, for instance our proposed abstraction level and subsequent trace comparison might improve our capabilities to detect equivalent mutants in the context of mutation testing. Since our instrumentation technique can be seen as an extension of *Javalanche*'s approach [51], we definitely see potential in this field.

As such, to encourage further research and ensure future development of *jHisteg* as well as the reproducibility of our results, we will make our tool available to the general public in the near future.

Bibliography

[1] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 246–256, 1990.

[2] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[3] Apache Ant. *http://ant.apache.org/*.

[4] Apache Maven. *http://maven.apache.org/*.

[5] Apache Subversion. *http://subversion.apache.org/*.

[6] Taweesup Apiwattanapong, Raúl A. Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. MATRIX: maintenance-oriented testing requirements identifier and examiner. In *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART 2006), 29-31 August 2006, Windsor, United Kingdom*, pages 137–146, 2006.

[7] ASM. *http://asm.ow2.org/*.

[8] D. Baldwin and F. Sayward. *Heuristics for Determining Equivalence of Program Mutations*. Department of Computer Science: Research report. Yale University, Department of Computer Science, 1979.

[9] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 85–103, 2007.

[10] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*, chapter An Introduction to Software Change Impact Analysis, pages 1–26. 1996.

[11] G. Booch. *Object Oriented Design: With Applications*. The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings Pub., 1991.

[12] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Inf.*, 18:31–45, 1982.

[13] Douglas Crockford. *https://github.com/douglascrockford/JSON-java*. JSON in Java.

[14] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for java. *Softw., Pract. Exper.*, 34(11):1025–1050, 2004.

[15] José Carlos Medeiros de Campos, Andrea Arcuri, Gordon Fraser, and Rui Filipe Lima Maranhão de Abreu. Continuous test generation: enhancing continuous integration with automated test generation. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 55–66, 2014.

[16] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2), March 1997.

[17] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[18] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419, 2011.

[19] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In Mauro Pezzè and Mark Harman, editors, *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 291–301. ACM, 2013.

[20] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Extending a search-based test generator with adaptive dynamic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 421–424, 2014.

[21] Git. *http://git-scm.com/*.

[22] GitHub Subjects. *http://www.evosuite.org/subjects/github-subjects/*.

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[24] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2):156–173, 1975.

[25] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. *SIGPLAN Not.*, 36(11):312–326, October 2001.

[26] William E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. Software Eng.*, 2(3):208–215, 1976.

[27] http-request on GitHub. *https://github.com/kevinsawicki/http-request*.

[28] Java. *docs.oracle.com/javase/7/docs/api/java/lang/Object.html*. Object class.

[29] Java Programming Language. *https://www.java.com/*.

[30] java.lang.Process. *http://docs.oracle.com/javase/7/docs/api/java/lang/Process.html*.

[31] JavaScript Object Notation. *http://www.json.org/*, json edition.

[32] JodaTime. *http://www.joda.org/joda-time/*.

[33] jsoup: Java HTML parser. *https://github.com/jhy/jsoup/*.

[34] JUnit Testing Framework. *http://junit.org/*.

[35] James C. King. A new approach to program testing. In *Programming Methodology, 4th Informatik Symposium, IBM Germany, Wildbad, September 25-27, 1974*, pages 278–290, 1974.

[36] Adam Kolawa and Dorota Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. Wiley, 2007. p. 426.

[37] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 308–318, 2003.

[38] Vladimir I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[39] Paul Dan Marinescu, Petr Hosek, and Cristian Cadar. Covrig: a framework for the analysis of code, test, and coverage evolution in real software. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 93–104, 2014.

[40] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for C programs with function pointers. *Autom. Softw. Eng.*, 11(1):7–26, 2004.

[41] Edward F. Miller, Jr. and R. A. Melton. Automated generation of testcase datasets. *SIGPLAN Not.*, 10(6):51–58, April 1975.

[42] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, 1963.

[43] Glenford J. Myers. *The art of software testing (2. ed.)*. Wiley, 2004.

[44] A. Jefferson Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Softw. Test., Verif. Reliab.*, 4(3):131–154, 1994.

[45] Hamilton L. R. Oliveira, Leonardo Gresta Paulino Murta, and Cláudia Werner. Odyssey-vcs: a flexible version control system for UML model elements. In *Proceedings of the 12th International Workshop on Software Configuration Management, SCM 2005, Lisbon, Portugal, September 5-6, 2005*, pages 1–16, 2005.

[46] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 815–816, 2007.

[47] Mauro Pezzè and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

[48] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Software Eng.*, 16(9):965–979, 1990.

[49] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[50] Raúl A. Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 218–227, 2008.

[51] David Schuler and Andreas Zeller. (un-)covering equivalent mutants. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 45–54. IEEE Computer Society, 2010.

[52] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272, 2005.

[53] Sina Shamshiri, Gordon Fraser, Phil McMinn, and Alessandro Orso. Search-based propagation of regression faults in automated regression testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*, pages 396–399, 2013.

[54] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. express: guided path exploration for efficient regression test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 1–11, 2011.

[55] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *RTI Technical Report*, 2002.

[56] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, pages 134–153, 2008.

[57] Mattia Vivanti, Andre Mis, Alessandra Gorla, and Gordon Fraser. Search-based data-flow test generation. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pages 370–379. IEEE, 2013.

[58] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the the application of revealing subdomains. *IEEE Trans. Software Eng.*, 6(3):236–246, 1980.

[59] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler Design - Syntactic and Semantic Analysis*. Springer, 2013.

[60] Zhihong Xu and Gregg Rothermel. Directed test suite augmentation. In Shahida Sulaiman and Noor Maizura Mohamad Noor, editors, *16th Asia-Pacific Software Engineering Conference, APSEC 2009, 1-3 December 2009, Batu Ferringhi, Penang, Malaysia*, pages 406–413. IEEE Computer Society, 2009.

[61] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.