The present work was submitted to
the Research Group
Software Construction

of the Faculty of Mathematics,
Computer Science, and
Natural Sciences

Master Thesis

Localizing Error-inducing Commits in CI Environments

presented by

Ricardo Hernández-Montoya

Aachen, April 13, 2016

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Andrej Dyck, M.Sc.

Statutory Declaration in Lieu of an Oath

The present translation is for your convenience only.
Only the German version is legally binding.

I hereby declare in lieu of an oath that I have completed the present Master Thesis's entitled

Localizing Error-inducing Commits in CI Environments

independently and without illegitimate assistance from third parties. I have use no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Official Notification

Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

I have read and understood the above official notification.

Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Master Thesis mit dem Titel

Localizing Error-inducing Commits in CI Environments

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, April 13, 2016 (Ricardo Hernández-Montoya)

Belehrung

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicher ung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen.

Aachen, April 13, 2016 (Ricardo Hernández-Montoya)

Acknowledgment

I would like express my deep gratitude towards Andrej Dyck for supervising this thesis. Through him and his willingness to share his domain knowledge and engage in discussions, many of the ideas reflected on this work arose, not to mention the valuable feedback that he constantly provided. Thanks to him I was immersed in this topic and I was able to strengthen my knowledge in this field.

Also, I would like to thank Prof. Dr. rer. nat. Horst Lichter for the opportunity of doing my thesis at the Software Construction Research Group and Prof. Dr. rer. nat. Bernhard Rumpe for agreeing to review my thesis.

Lastly, I thank my family for the constant support they have given me along my studies.

Ricardo Hernández-Montoya

Abstract

Nowadays, the practices around *Continuous Integration (CI)* have gained significant traction as project codebases are being impacted by a larger number of developers each time, and the need for a consistent and stable project remains a priority. In this regard, CI considers conducting a testing phase for each contribution made to a codebase. Despite of providing a higher level of confidence to the changes done, the testing phase can represent a time-consuming task. Also, it can become challenging to identify the contribution responsible for a failed test if multiple of them are being evaluated. Large IT organizations solve this by leveraging parallel jobs to validate each contribution, while still requiring to execute the test sequence on each of them.

This thesis conceptualizes strategies that aid to identify contributions —i.e. commits made to a software repository—that led to a failed test. We call those *error-inducing commits (EICs)*. To achieve this goal, existing code analysis techniques are explored and exploited to deduce the commits responsible for a test failure by associating the past code changes to the sections exercized by the test. In addition, this work provides a schema of properties related to the extend at which a tracing strategy discovers EIC candidates, which is used to categorize the conceived strategies. Furthermore, a tracing strategy is realized and its behavior is demonstrated in a set of common development scenarios. Towards the end of this work, suggestions are made for the further conception and realization of tracing strategies.

Contents

List of Tables

List of Figures

List of Source Codes

1 Introduction

Software development is an activity that can involve a large number of developers who jointly contribute to the *codebase* of a project. Along the history of this activity, diverse approaches have surged to encompass it in order to improve the productivity, efficiency, and ensure the quality of the deliveries. These approaches have taken many forms, including: process models, philosophies, tools, etc.

Continuous Integration (CI) is a practice that has gained a lot of traction in the recent years. CI aims to ensure the consistency and stability of the codebase by assisting in the integration of each developer's contribution into the project codebase. CI relies on *test automation*, the process of conducting tests directly after a change to the codebase has been detected with the purpose of ensuring the healthiness of the project.

As software projects increase their dimensions, they demand a larger number of participants working simultaneously. In this scenario, besides assisting the project development, the assessment of each code contribution can represent a big overhead and, ultimately, a bottleneck to the development process if it does not keep up with the pace of the development.

Nowadays, large IT companies and projects, such as Google[Kum10], Facebook[FFB13, Fac15], and OpenStack[Ope12], have already the demand that has pushed them to develop their own infrastructure to handle their own software development workflow which consists, basically, in assessing each contribution independently and determining if it can be safely integrated to the project by executing automated tests. However, besides their experiences and solutions, there is limited research that focuses on this issue.

Despite of this, there exists enough literature in the field of *regression testing* that could aid to identify, in the event of a test execution failure, the contribution that has introduced the error in the codebase and, hence, procure a faster build fixing and avoid holding the development process. One of the concerns of the regression testing research is to improve the efficiency of executing tests to discover errors after code changes, for which it relies on *software dependency* techniques that help to identify which tests exercise the modified code changes. Such techniques can be reused in order to conceive techniques that could help to link the code changes that affected a codebase and the failed tests that evidence the presence of an error.

Contributions

The aim of this thesis is to conceive strategies that help to localize *error-inducing commit (EIC)* in a *Continuous Integration (CI)* environment. In this regard, the contributions of this thesis are outlined next:

1. An overview of the techniques used in the field of regression testing, specifically in *code dependency analysis*, is provided. Additionally, this overview covers previous work that is related to the concept of discovering EICs although not specifically associated with a CI environment.

2. The concept of tracing EICs and a set of strategies that accomplish it are introduced. These strategies are, additionally, compared between them and categorized according to a properties schema that, together with the strategies, was conceived in this work.

3. Beyond conceptualizing the strategies, one strategy was realized by extending a framework for regression testing. The behavior of the implemented strategy is shown in diverse demo cases that exemplify common development scenarios. The results observed in this scenarios are discussed.

4. Furthermore, this work concludes with further work suggestions and implementation ideas.

Structure

This work starts, in chapter 2, with an overview of the work related to regression testing, discovering EICs, as well as, practices for avoiding EICs. Chapter 3 introduces the concept of tracing EICs, the tracing strategies, and their evaluation. Chapter 4 introduces the framework for regression testing which was extended for implementing the tracing strategy together with the technologies involved and, in addition, describes the extensions needed to support EIC-tracing. In chapter 5 specific details related to the implementation of the strategy are given. Next, chapter 6 tackles the evaluation of the strategy implemented with the help of demo cases. Later, chapter 7 discusses the strategies in terms of the properties schema introduced in this work and addresses their strengths and weaknesses. Lastly, chapter 8, covers final conclusions and points to further work directions.

2 Related Work

Contents

2.1 Regression Testing

Software regression testing is a process where developers continuously verify the behavior of software as they maintain the code. This is done using a test suite, which is steadily executed with the hope of discovering errors and gain confidence as the code is modified. [GHK+01] Running regression tests can represent a very expensive task and a big overhead in the software development process. Therefore, current research focuses on improving techniques for test selection and prioritization which, ultimately, aim to reduce the cost of running regression tests while maximizing the detection of faults.

2.1.1 Change Impact Analysis

Regression Testing can benefit from *Change Impact Analysis (CIA)*, a set of techniques that aim to identify the software entities that may have been affected after a change between two versions. [Leh11b] With the help of CIA, regression testing can determine which tests to run and how to prioritize them, in order to exercise the changes that are more likely to contain errors and discard the tests that were not impacted by the changes.

There have been many studies in the area of CIA and efforts have been made to categorize them. One of the most recent and complete classifications can be found in [Leh11b]. According to this work, the studies rely on a set of distinct techniques:

- Program slicing.
- Call graphs.
- Execution Traces.
- Program Dependency Graphs.
- Message Dependency Graphs.
- Traceability.
- Explicit Rules.
- Information Retrieval.
- Probabilistic Models.
- HistoryMining.

The techniques shown above represent quite diverse approaches and involve different software artifacts, including source code, models, or a combination of them. Relevant to this thesis, we can identify a set of techniques that rely on a source code-based *dependency analysis*, namely, *Program slicing*, *Call graphs*, *Execution Traces*, *Program Dependency Graphs*, and *Message Dependency Graphs*. Dependency Analysis will be covered thoroughly next.

2.1.2 Source Code Dependency Analysis

Dependency analysis based on source code can help to determine whether a statement has been impacted after certain syntactic change. The techniques that achieve this analysis can be further classified in: *static analysis* and *dynamic analysis*.

Static analysis techniques don't rely on a specific execution path and hence approximate on the *potentially* affected execution paths and how the semantic effects are propagated. According to Böhme et al. [BRO13], with the help of static analysis, it is possible to: determine which statements are definitely not affected by a change or just under a certain probability, whether a set of changes *semantically interfere* between them, and which statements depend syntactically from a changed statement. Techniques that can provide this analysis include: call graphs, program slicing, and program dependency graphs.

Dynamic analysis techniques, on the other hand, rely on a specific execution path which is constructed using a defined program input. Based on the execution path, it is possible to precisely determine the statements that were affected by the exercised changes, if the changes are propagated to the output, or if different subsets of changes interact between them. [BRO13]

Call graph-based techniques

Techniques that rely on this method perform a static analysis to study the call-relations inside a source code. Once the relationships have been collected into a so-called *call graph*, it is possible to determine affected methods if they call a modified method. This can be computed using transitive closures.[Leh15] A comprehensive list of techniques implementing this method has been compiled by Lehnert [Leh11a].

Program slicing-based techniques

A program *slice* consists on a subset of the program statements which are related to a specific program element (*slice criterion*), e.g., a variable or a control statement. The slice can be computed in a *forward* or a *backward* manner. A forward slice reflects the statements that depend on the criterion, including the output if it is reachable; while the backward slice identifies the statements that influenced the value of the criterion.

Although the slicing technique is usually catalogued as a static analysis technique, there exists *static* slicing, as well as, *dynamic* slicing. The objective of the analysis may play a role on which type of slicing to use. While the static slicing is usually employed to

understand a program, the dynamic slicing is usually more suitable for tracing an error in a program (Debugging).[Lig09]

2.2 Discovering problem origins through changes history

Efforts have been done in the past to examine software changes history and identify *problems* which a developer may have inadvertently introduced in a software while performing maintenance, implementing new features, or fixing bugs. This has been addressed before using the terms *bug-introducing change* and *fix-inducing change* while referring to the same concept. Examples of these efforts can be found in works of Sliwerski et al. [ŚZZ05], Kim et al. [KZPJ06], Aversano et al. [ACD07], Williams and Spacco [WS08], Sinha et al. [SSR10], Wu et al. [WZKC11].

The main target of these studies is to establish a link between *bug-fixing changes*—the changes that introduce a fix—and the bug-introducing changes, while relying on bug databases (e.g., bug-tracking systems) or bug reports. This is achieved using different approaches, such as: text-based source code comparisons [ŚZZ05, KZPJ06, ACD07, WS08], analyzing program dependences [SSR10], or features related to the changes logs [WZKC11].

It is worth mentioning that this research has been motivated mainly by the interest of knowing properties of bug-introducing changes and discovering common change patterns that are susceptible of introducing bugs. Although this is outside the field of CI, it is relevant to understand the approaches employed to link past code changes along code versions.

2.3 Related tools

Currently, there are tools that aim to maintain a healthy build system by avoiding changes that could corrupt the codebase of a project.

Two different known approaches will be described below accompanied by the tools that implement them.

2.3.1 Gated commit

Also called *gated check-in*, is a software build pattern which enforces code changes verification separately and in a isolated environment before integrating them into the project's main line of work.[Osh13]

A tool that provides an automated version of this pattern is OpenStack Zuul, a gating system project which makes it possible to verify a set of code changes received during a period of time by running separate build jobs in parallel and integrating them into the project only if they passed a set of tests. In addition, Zuul is capable of handling dependencies between code changes. It can hold certain change if it doesn't comply the tests, while processing the rest of the changes set.[Ope12] Being able to test individual code changes in parallel enables a significant improvement in feedback response time,

which could be otherwise impossible considering that in large projects a test execution could take several hours and a codebase may be impacted by a large number of changes in a single day.

2.3.2 Binary approach

Certain tools have implemented a common binary-like method while aiming to discover code commits that break a project build. This approach consists on examining, after the presence of an error has been evidenced, the previous commits history in a binary manner in order to determine which commit is responsible of introducing the fault.

An example of such feature is the `bisect` command, featured by the Git source code management system.[Git12] This command offers two possible modes of operation: *manual* and *automatic*. Both modes allow to set an initial and ending commit to perform the analysis. However, while the former requires the user to execute an operation (e.g., test execution) on each commit and mark it as *good* or *bad*, the latter is configured using a script, which is triggered on each commit visit, such operation is conceived for longer bisect runs that difficult a manual execution.

A common workflow used to identify the origin of a flaw in a repository is shown in figure 2.1. Based on the evidence that a flaw exists, the inspection can start with generating a test case which evidences the flaw. Then, a script can be generated to execute the test case on a certain code version. Finally, with the help of `bisect`, a long inspection run (usually overnight) can be executed between the last known working version and the most recent one, which will determine the faulty commit. In combination with additional tools, it is possible to extract information about the commit author, notify him, and, in addition, create an issue item, which can be tracked in a issue management system.

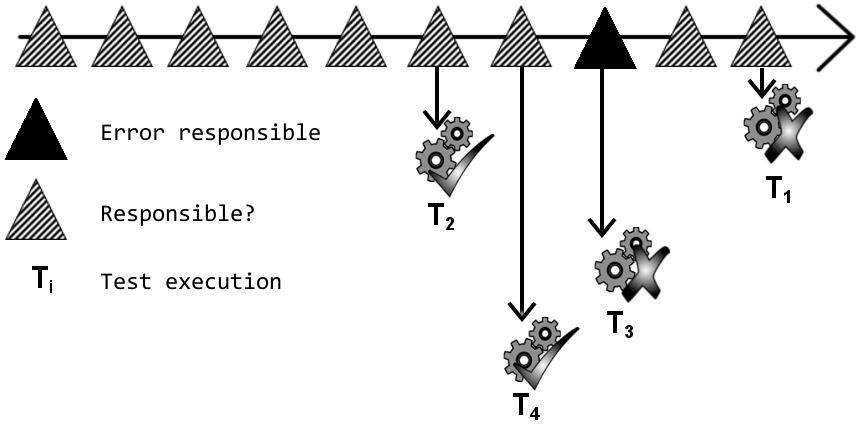


Figure 2.1: Binary approach to identify commits responsible of breaking a project's build.

Bisect is a tool popularly used by developers and QA staff. Because of this, other platforms mirror the behavior of `git bisect`, such as: a similar command for the

management system called Mercurial [Mer] and a plugin for the continuous integration tool TeamCity.[Tep15]

The binary approach of finding faulty commits has been adopted further than the examples mentioned before. Google is an example of a company that implemented such approach within their test infrastructure, their implementation gives them the opportunity to localize a *breaking* commit in their large and highly-active integration pipeline.[Kum10]

Facebook is another example of a company that implements this approach. Facebook uses automated bisect within their in-house developed continuous integration system called *Sandcastle*. This system helps the organization to handle the integration of each developer's commit into the main line of work without holding their work and providing them rapid feedback in case of failures.[Fac15]

3 Concept of Tracing Error-inducing Commits

Contents

This chapter covers one of the main contributions of this thesis: the foundations of the concept of EICs and the tracing strategies. Before these concepts are introduced, a brief background is given, followed by the EIC concept complemented with a properties schema, an output report structure and ideas regarding the evaluation metrics. Next, each of the strategies will be described.

3.1 Background

Before the topic of tracing EICs is covered and the tracing strategies are explained, this section introduces the concepts behind those topics, which may result helpful in order to understand them. This concepts include: a brief comprehension of Version Control Systems (VCS), the workflow description and common practices of a Continuous Integration environment, and an overview of common testing frameworks.

3.1.1 Version Control Systems

As a software project is developed, it is important to keep track of the source code changes (and the relevant resources) in a repository which is available to all team members. For this reason, modern Version Control Systems (VCS) provide guarantees such as the ones found in a database; namely integrity, shareability, and integrability.[Pre10]

Key features of a VCS are:

- Storing all source files, documents, libraries necessary to build the project, including all of their different versions.
- Providing each of the team members an isolated line of work into which they can store their individual contributions (*commits*).

- Providing the capability of integrating separate lines of work into a single one.
- Possibility of recovering contextual information of each of the contributions made by the team members, such as the author, time and the details of the changes involved.

Three of the most popular VCS nowadays are: Git[Git16], Subversion (SVN)[Apa15]; and Mercurial[Mer16]. Although their main purpose is to store resources in a similar fashion as in a *filesystem*, a key difference between them is the repository model they follow. On the one hand, SVN follows a traditional *central repository model*, where the project changes are managed in a single place and each of the members keeps a working copy in their local environment eventually committing their changes back to the central repository. On the other hand, Git and Mercurial follow a modern *distributed repository model*, where each of the participants hosts their own repository and it is updated from their peers. Despite of their order of appearance, SVN may present advantages over Git and Mercurial, and viceversa. This comparison is, however, beyond the scope of this introduction.

3.1.2 Continuous Integration

Continuous Integration (CI) is the evolution of software integration, achieved as a result of software involving a larger number of team members and consisting of a bigger number of artifacts and dependencies. Although software integration was practiced already in previous software development methodologies, it was until Extreme Programming (XP) and Agile appeared that practitioners started considering integrating several times a day.

According to CI evangelists, i.e., Duvall et al. and Fowler, one organization can embrace this methodology by following several principles[DMG07, Fow06]:

- Keeping all software assets in a single central source repository.
- Automating builds in order to save time in repetitive tasks such as compiling, performing inspections and running tests.
- Building self-testable source and implementing the execution of automated tests into the integration cycle.
- Making frequent commits in order to, not only facilitate code merges, but also reduce the probability of introducing errors, since each commit represents a smaller chunk of code.
- Commit-triggered integration process (either automatically or manually) should rapidly inform the committer if the build failed.
- Broken builds should be notified immediately in order to avoid delaying the integration pipeline. Since the commits represent ideally small changes, this should be easy to fix.

- Test the application in clones of the production environment. This is done in order to avoid assumptions that could arise once the application is deployed.

It is possible to list some clear benefits of adopting this methodology that have attracted many organizations to embrace it, such as: Mitigating risks, by discovering errors in an earlier stage. Process automation can avoid errors and save time of doing repetitive tasks manually, which can help team members to focus in development. Doing continuous integration of code commits allows a project's codebase to be deployable at any time and release new version in a more frequent basis, this is however, the focus of another philosophy called *Continuous Delivery* and involves, between other things, configuring infrastructure automatically. Lastly, have a better project visibility, accompanied by quality metrics, defect rates, speed of feature completion, which could help to take important decisions regarding the project.

This thesis focuses on conceptualizing strategies that could help to discover commits that introduced errors into a codebase. This could be of great relevance in CI, as these strategies could help to recover from an interrupted integration cycle once a build broke and keep the integration process continuous.

3.1.3 xUnit

The term *xUnit* corresponds to a family of *Test Automation Frameworks* that follow the same practices of test definitions, yet implemented in different programming languages[Fow11]. The name originates from JUnit, the first member of this family for the Java programming language which, after being made known, was quickly adopted and fostered the implementation of the same concept in different technologies.

xUnit frameworks come in many flavors and there is a reason for that (besides technology compatibility); namely, that developers are frequently in charge of developing tests together with the software in order to verify it. Therefore, it is convenient to be able to write tests in the same programming language used to develop.

Test pattern

Following the xUnit pattern, each of the test cases is implemented using a *test method*[Bec02, Mes06]. This method exercises the code section targeted by the test case and, once completed, the expected output is evaluated against the real output using *assertionss*, statements whose solely purpose is to determine if the *system under test (SUT)* behaved correctly under the test case. In case the test fails, the result is usually accompanied by an *assertion message* which helps to identify the reason of misbehavior.

Since an object may be the target of several test cases, these test cases are grouped into *test classes*, where each test class includes a *fixture*, that is, the common objects required by all test cases.

One of the main goals of xUnit frameworks is to achieve test *automation*, for this reason, their test classes implement a common *four-phase process*. During this process, a sequence of four steps is executed, consisting in:

- First, during the *set up phase*, configuring the fixture as the environment where the SUT would normally perform.
- Then, executing the test method and performing the interaction with the SUT.
- After that, evaluating the obtained interaction outcome.
- Finally, during the *tear down*, bringing back the environment to its original state before the test.

By following this process, it is not only easier to identify the test case-specific conditions, but also the test cases can be written and executed in a more efficient way, since some elements may not be duplicated or set up redundantly.

Once all test classes are in place, it is a common practice to run all of them at once during a verification step. In order to do this, the test classes are grouped into a *Test Suite*, which can easily by identified and executed by an actor called *Test Runner*.

3.2 Error-inducing commits

Before proceeding to introduce the conceived tracing strategies, it is important to define the target of our discovery efforts, namely, the *error-inducing commits (EICs)*.

First, a *commit* should be understood as a contribution made by a project team member to the main repository of the project. A commit can include different types of contributions, such as source code, documentation, configuration files, third-party libraries, etc; which are required for the project and can coexist in the repository. However, for the purpose of this thesis, when mentioning the term *commit* we focus on (and we will refer to from now on) source code contributions.

The purpose of a commit, i.e., code contribution, is to enhance a software project by implementing a new feature, fixing an existing bug or maintaining the current codebase. This enhancement is usually followed by a behavior verification carried out by *regression testing*. A failure to comply a test execution (assuming a well-written test) will represent an error that was introduced in the codebase. This gives us enough elements to formulate the concept of EICs (cf. figure 3.1) as *source code contributions that have introduced an error in the project codebase, whose existence has been evidenced by a failed test execution.*

Discovering EICs is the focus of our efforts. This can represent a special challenge if the codebase has been altered by several commits since the last successful integration. The main task involves now, associating each of the commit's changes to the code sections reached by the tests, which acknowledge the presence of an error. This, however, will be covered in detail in the following sections.

3.3 Tracing strategies fundamentals

This section is concerned about introducing the concept of EIC-tracing strategies. These strategies build upon the fact that the project under development has a defined

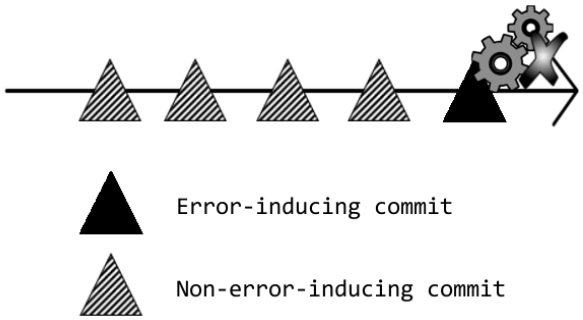


Figure 3.1: Representation of an error-inducing commit, where the presence of an error is evidenced by a failed test execution.

automated test suite, which is continuously extended along the development cycle following *Test-driven development (TDD)* practices and, also, that the project evolution is managed by a *version control system (VCS)*.

During a normal workflow, the project will integrate the contributions of the developer team and will execute the test suite continuously. In case the test execution happens to fail, the strategies will take action and will focus on localizing the EIC, which may yield a number of candidates.

Next, fundamentals concerning the common strategies input, properties, output reporting and evaluation metrics will be presented.

3.3.1 Strategies input

Before presenting specific details of the tracing strategies, three common input elements that provide the basic information needed to perform the tracing will be described. These elements comprise: gathering the test execution coverage, fetching the previous test history; and collecting the code changes history.

Test code linking

In the context of this thesis, the presence of errors is evidenced, essentially, through failed test cases. Hence, if the focus of the strategy is to localize the introduction of an error into the project's codebase, it is relevant to link the sections of code exercized by tests, this can be achieved using code coverage or call-relationships. In regards to code coverage, it is important to know which code was covered on a *line-level* detail, since a previous change may have occurred in a single line and, therefore, restricting ourselves to *method* or *branch* coverage would not be sufficient. Regarding call-relationships, code analysis techniques can help to determine the call dependencies on a method level (see section 3.6).

The outcome of this step would be a collection of covered source lines for each of the source classes touched by a single test or a method call-graph.

Previous test history

In order to start narrowing down which code commit may have introduced an error, it is important to have an initial version to start off the analysis. This version corresponds to the last code version, whose test suite was successfully executed, denoted *succesfully-tested codebase version (STCV)*. The STCV may change during the project's lifecycle. Therefore, it is important to keep track of the past test execution history together with the id of the code version being tested. Since versions are uniquely identified in a VCS, it is simple to use it to fetch the following most recent versions.

Changes history retrieval

With the information we have about previously tested code versions (specifically the last STCV), we can perform a query to find out which files were recently modified. This query aims to relate the code modifications that took place to the corresponding commit. This information is then later used to perform strategy-specific operations in order to determine if a commit is a EIC-candidate.

3.3.2 Strategy properties

By now, the different strategies and their approaches have been introduced. However, if it is required to employ them, understanding their approach might not be enough. Furthermore, it would be important to know the guarantees that they provide and depending on those, one could decide which one to opt for. In this section, a set of properties are introduced which can be used to categorize a tracing strategy. In a later discussion section (see section 7.1) each strategy will be catalogued according to this schema. This set of strategies has its origin in a work by Rothermel and Harrold[RH96], where they were part of a framework used to analyze regression test selection techniques. In the context of that work, the properties described the extent at which the strategies included and avoided modification-revealing tests.

Properties schema

Below are presented the properties that will help to categorize the strategies:

Safe strategy. Strategy which discovers all possible EICs. Though there could be false-positives, this technique include among its candidates all real EICs.

Precise strategy. Strategy which avoids all non-error-inducing commits. This technique may not deliver all EICs, however, all candidates that it detects are real EICs. This means, in turn, that there are no false-negatives among its results.

Reliable strategy. Strategy which delivers at least one possible EIC. This technique may not deliver all EICs nor it can be assured that all candidates are EICs. However, this strategy guarantees to deliver one EIC among the candidates.

Figure 3.2 allows to visualize the possible combinations of these properties. As shown in the diagram, any strategy that delivers at least a valid EIC can be catalogued as a *reliable* strategy. If a strategy manages to discover all EICs, it is considered a *safe* strategy and, if it also avoids false-positives, can be considered *precise* as well. Moreover, if a strategy delivers valid EICs, while avoiding false-positives, it is considered *precise*. Finally, a strategy that does not deliver valid EICs does not fall into any of these categories.

	Safe	Precise	Reliable
• - - •	X	X	X
— —	X	-	X
— - —	-	X	X
———	-	-	-

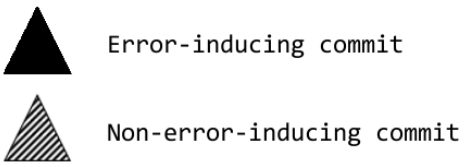


Figure 3.2: Properties schema that can categorize a strategy based on the capability of identifying valid EICs and avoiding false-positives.

3.3.3 Tracing output reporting

In the previous sections, the conceived strategies as well as the properties to categorize them were presented. However, still an important part of the concept of tracing EICs is to present the gathered information or even determine which information is available and helpful.

Below, the content of a tracing output report is presented accompanied by the description of each parameter:

EIC id. Codebase version id of the detected EIC. Id that helps to uniquely identify a commit in a VCS.

Commit date. Commit creation date.

Commit author. Person responsible for creating the commit in the VCS.

Author contact. Author's contact medium, usually the e-mail that the author uses to authenticate with the VCS. Can be potentially used to trigger an immediate

notification directly to the author and urge an immediate build fix, one of the main objectives pursued by CI.

Current code version. The id that identifies the current version being tested.

Last successful code version. The id that identifies the last STCV that was used as a starting point to execute the strategy.

Tracing strategy. The name of the strategy employed.

No. EICs found. Number of EIC candidates found.

Total commits analyzed. Total number of commits found between the current tested code version and the last STCV. The number of commits analyzed during the strategy execution.

Ratio. Ratio of EIC candidates and the total of commits analyzed.

The output report shown before represents the result of a single strategy execution. Although observing the outcome of a single strategy execution may not be very significant, collecting the results of several executions and comparing it against other strategies, may result in interesting statistics and metrics could be deduced from them. These will be introduced in the next section.

3.3.4 Strategies evaluation

Before, the information that can be obtained from a tracing strategy execution was already presented. Now, it will be presented how the gathered information could be examined and converted into insight that would better show the general performance of each of the them.

Evaluation metrics

Below is a list of the metrics that could be obtained from a series of executions of the strategies:

Time. Comparison of the runtime of the different approaches.

Result set size. Percentage of the delivered commits out of the analyzed set.

Precission. Extent at which the approaches deliver valid EICs.

Inclusiveness. Percentage of the result that represent valid EICs.

Error-rate. Percentage of the result that represent non-EICs.

An example evaluation result is shown in table 3.1 including the execution of three hypothetical strategies where the focus of the evaluation were 10 commits belonging to the history of certain project.

Parameter	Strategy 1	Strategy 2	Strategy 3	$\overline{X}$
Running time (s)	11	12	10	11
EIC candidates	4	3	5	-
Real EICs	1	2	1	-
Non EICs	3	1	4	-
Results (%)	40	30	50	40
Inclusiveness	1/4	2/3	1/5	-
Error-rate	3/4	1/3	4/5	-
Precision	1/3	2/2	1/1	-

Table 3.1: Example evaluation results from three hypothetical strategy executions.

3.4 Statement-test coverage-based strategy

The *statement-test coverage-based strategy (STC)* is the first strategy that will be presented. The main principle behind STC is the usage of the failed test coverage information, in terms of source code lines, and deduction of which commit could have introduced the error based on the modified source lines.

As mentioned before, the strategy relies on three elements in order to be able to deduce the EIC-candidates: the code coverage of the failed test, a record about the last STCV; and the changes history since the last STCV. These elements should be available before executing the core algorithm of this strategy. Then, the information provided by these elements will be combined and the code coverage will be matched with the changes history in order to deduce the EIC-candidates (cf. figure 3.3).

A special consideration during the retrieval of the changes is that, as the source of the code files evolves, the covered source lines counting may not exactly correspond to those in previous code versions. For this reason, an approach of history context has to be adopted, it will be called *DeltaSpots*. This approach consists of keeping track of the lines added or removed during the code evolution in order to be able to perform a line-matching in the core step of the strategy. This process takes place during the changes retrieval and has to be done starting from the most recent code version (version on which the tests were executed) back to the oldest code version being analyzed. This step is sketched in figure 3.4.

Code change-coverage matching

The core step of the strategy combines the information obtained after the common steps. At this stage, all the necessary information is available: a collection of covered source lines obtained during the *test code coverage* and, also, the specific modified lines for each file that was identified as modified in the past commits, information which was obtained *changes history retrieval*.

At this step, it is fairly straightforward to deduce the EIC candidates. Following the below algorithm:

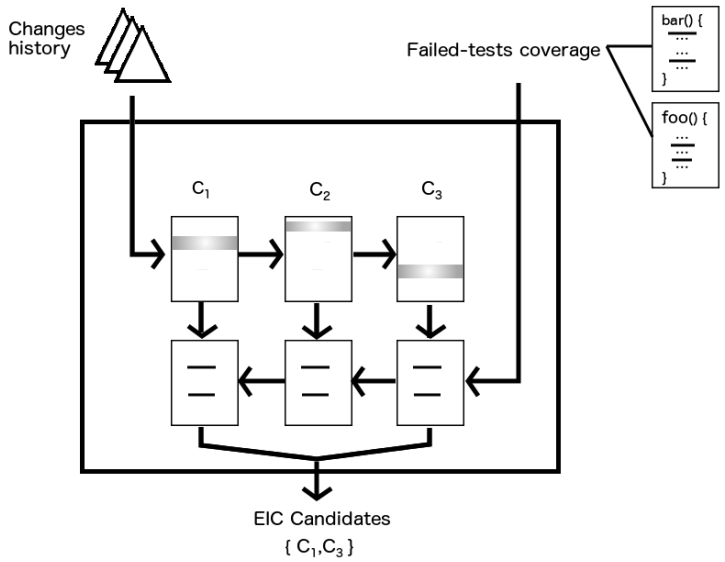


Figure 3.3: Concept of statement-test coverage-based strategy, featuring the covered-modified-line matching.

1. After the text execution, consider all failed tests.
2. For each of them, consider the project classes that the test covered.
3. For each of the project classes, check if it was modified in the *change history*.
4. If changed, perform a *line-matching* operation between the test coverage and the modified lines, for each of the commit changes that were identified.
5. For each *coverage-modification-line* match, extract the responsible commit.
6. Finally, all responsible commits are collected and reported as EIC candidates.

3.5 Slicing-based strategy

The *program slicing-based strategy (PS)* will be now presented. This strategy builds upon a technique called *program slicing*, a powerful technique used for software dependency analysis. This strategy aims to relate the code sections covered by a failed test to the code statements that a code change may have impacted to in order to determine EIC candidates. This approach will be explained below after a brief introduction of the program slicing concept.

3.5.1 Background: Program Slicing

Section 2.1.2 introduced the concept of program slicing as it is used for regression testing techniques. In that section, two types of slicing were mentioned: static and dynamic

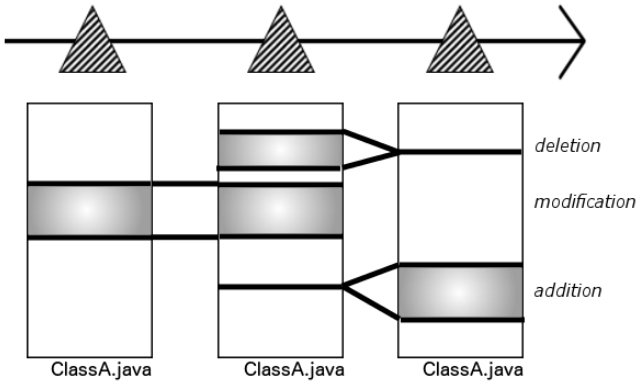


Figure 3.4: Representation of delta spots, which allow to track code lines insertions and removals along the changes history.

slicing, where static slicing is commonly used to understand a program, while dynamic slicing is better suited to trace back an error (Debugging process). Although the use of this technique in this work is motivated by the localization of an error in the codebase, the primarily focus of the tracing strategy is to understand the changes that the code has suffered in the past commits, hence the static approach is used.

A forward slice is built with the help of a *Control-Flow Graph (CFG)*, a graph whose nodes represent code statements and the vertices are arranged in such a way that describe the possible execution paths of a program. An example of this graph is shown in figure 3.5b corresponding to the code snippet of source code 3.5a. Having computed the CFG, it is possible to identify the statements that depend on a certain statement, the *slice criterion*. A common way to represent a slice is by using a diagram. The forward slice corresponding to statement `sum` is shown in figure 3.5c.

3.5.2 Tracing strategy

Having introduced the concept of program slicing, we can describe the approach followed in the PS strategy (see figure 3.6). Similarly to the past strategy, STC, this strategy also relies on three elements: the code coverage of the failed test, a record about the last STCV; and changes history since the last STCV.

Regarding the changes history, differently to the last strategy, PS requires to know exactly which statements were modified in the commit history, instead of the source lines of the corresponding source files as done in the previous approach. The reason for this is explained next.

3.5.3 Changes history retrieval

The objective of this step is to discover which statements were modified along the changes history and compute slices for each of the statements modified in the source code. These

slices are associated with the class file where they occurred and the commit where they were detected. By using forward slices it is possible to keep track of the statements that could have been affected by this change.

3.5.4 Commit blaming

At this stage, the information about the statements covered by the failed tests is used to match the code sections that a commit's changes may have influenced (see figure 3.6b).

The process used to deduce the EIC candidates follows the steps below:

1. After the text execution, consider all failed tests.
2. For each of them, consider the project classes covered by the test.
3. For each of the project classes, look up for the class in the slices collection gathered during the change history retrieval.
4. If found, proceed with looking up the covered statements of the corresponding class in the statement slice traces of the change history.
5. If any of the statements match, mark the responsible commit (whose slice belong to) as a EIC candidate.
6. Finally, after the coverage of failed tests have been analyzed, all the EIC candidates identified during the strategy run are reported.

3.6 Atomic changes-based strategy

The strategy presented here is based on the work of Ren et al. [RST+04] , which consists on a change impact analysis tool for *Java*. This tool builds upon the analysis of *atomic changes*, a set of operations that "capture the source code modifications at a semantic level that are amenable to analysis." [RST+04] A detailed explanation of this concept and how they are used in the prototype of Ren et al. follows.

3.6.1 Background: Call-graphs

In section 2.1.2, it was mentioned that call graph models is a powerful technique used to gather dependency relations inside source code. It is a technique that has been exploited in terms of CIA to understand the software behavior impacts between code versions.

3.6.2 Background: Atomic changes

In their work, Ren et al. [RST+04], present a prototype of change impact analysis for Java that differences from other tools developed in two factors: first, besides identifying the differences of two program versions, the tool also reports the change impact in terms of the regression tests whose execution behavior may have been impacted by the changes;

second, it doesn't compute the program differences by comparing control flow graphs, instead it relies on the analysis of atomic changes.

As mentioned before, atomic changes capture source code modifications at a semantic level. The modifications types handled in the prototype of Ren et al. [RST+04] include:

- Added classes (**AC**)
- Deleted classes (**DC**)
- Added methods (**AM**)
- Deleted methods (**DM**)
- Changed methods (**CM**)
- Added fields (**AF**)
- Deleted fields (**DF**)
- Lookup changes (**LC**)

As the differences between the versions are identified, each change is interpreted into the corresponding atomic change. Using the atomic changes, syntactic dependences can be deduced and, based on these, it can be determined if the behavior of a test was impacted by the change.

Figure 3.7 depicts an example of a change impact analysis as carried out in the work of Ren et al. The source code that is subject of analysis is shown 3.7a, where the changes identified are highlighted by boxes. Each of these changes are interpreted into atomic changes in 3.7b and the dependencies between the elements are indicated by the arrows. It is important to notice that a change to a method is split into the *added method (AM)* and *changed method (CM)* atomic changes. Finally, the impact to a test method can be identified by its *call graph*. If one of the methods contained in the call graph matches any of the atomic changes, then, it can be deduced that the test behavior has been impacted by the change. It is worth mentioning that the presence of other atomic changes could be already discarded, which makes this approach more amenable.

3.6.3 Tracing strategy

Now that the concept of atomic changes has been introduced, we can elaborate on how it is used in the tracing strategy (see figure 3.8). We should recall that the process of localizing EICs starts when a failed test has been detected. At this point, the tracing strategies will be triggered. In the case of *atomic changes-based strategy (ACS)*, three main elements are needed to proceed with the tracing: a record about the last STCV, the detected changes in terms of atomic changes, and the test method-call information.

The process of gathering the code changes in the VCS starts by comparing from the commit corresponding to the STCV up to the most recent commit. For each of the changes it is determined if it represents a method modification, a field added, or a look up change. The result of the previous step is then translated into an atomic change as seen in the previous section. Before delivering the result of the code change gathering, the accumulated atomic changes are associated with the code version where it was detected at.

Besides gathering the code changes information, another important step involves building call graphs for each of the failed tests, which results in a similar output as

in figure 3.7c.

Finally, after the code changes, in form of atomic changes, and the test method-call relations are available, it is possible to associate the failed tests with the atomic change that may have impacted it (if any) and extract the responsible commit (see figure 3.8b). The set of of the responsible commits are then delivered as output.

3.7 Rerun tests on all failed commits strategy

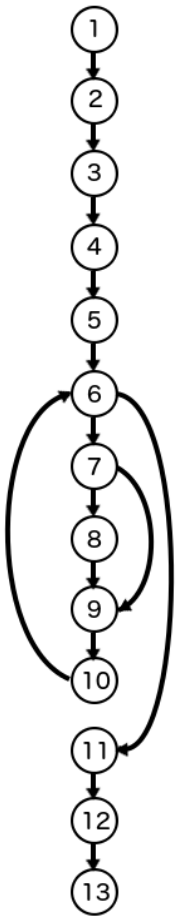
The *rerun-all-failed-tests strategy (RAFT)* involves re-running the previously failed tests on all past commits. Although this is a naive approach with clear processing overhead, it does guarantee that the EIC candidates are indeed the ones we would expect to obtain, discarding hence the presence of false-positives.

The figure 3.9 illustrates the involved approach. In the figure, we can observe that, in the event of a failed test execution, the strategy will proceed to query the test execution history to find the code version corresponding to the last STCV. Once obtained, the strategy will start executing the test suite on each following code version progressively. Every code version that happens to fail the a test case for the first time will be logged as EIC candidate. The strategy will stop, however, until all commits *responsible* for the failed tests are detected or the strategy has reached the latest commit (our originally tested code base version). This is done in order to make the best effort discovering the probable EIC.

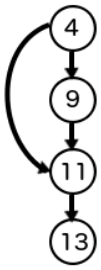
A special consideration needs to be done, however, when executing the test suite on previous code versions as the test suite itself may have also evolved along them. For this reason, it is needed to keep track of the test cases belonging to each version.

```
1  input(n);
2  input(a);
3  max := 0;
4  sum := 0;
5  i := 1;
6  WHILE i <= n DO
7      IF max < a[i] THEN
8          max := a[i];
9      sum := sum + a[i];
10     i := i + 1;
11 avr := sum / n;
12 output(max);
13 output(avr);
```

(a) Code snippet.

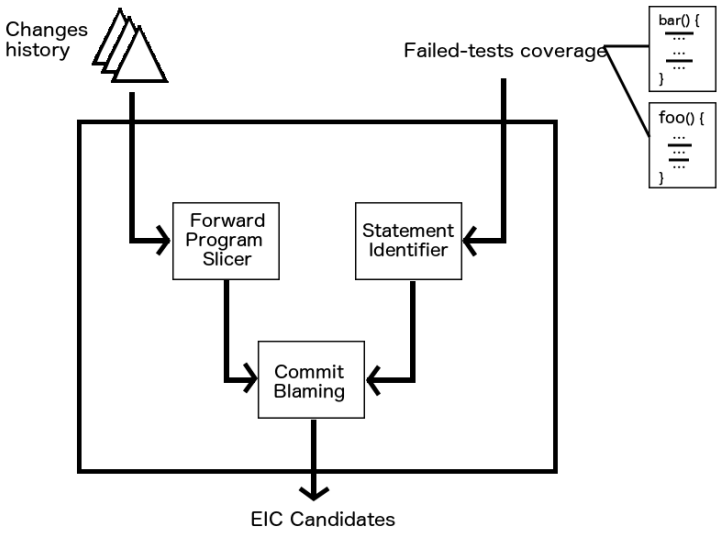


(b) CFG.

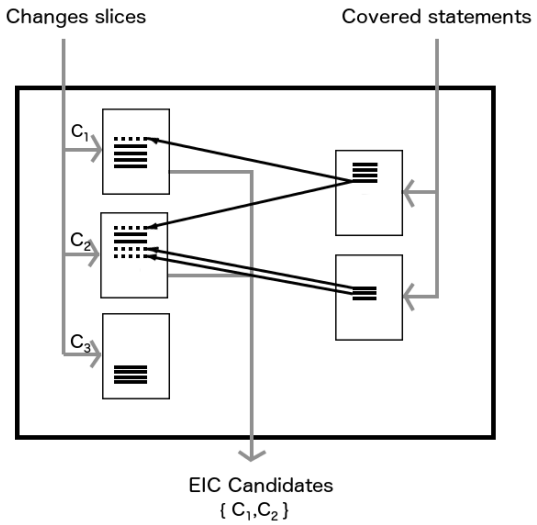


(c) Forward slice.

Figure 3.5: Example of a slicing operation showing: a) a code snippet with control statements, b) the corresponding CFG, and c) the resulting forward slice using variable `sum` as slice criterion. Adapted from [Lig09]



(a) Block diagram.

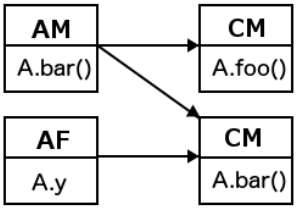


(b) Commit blaming.

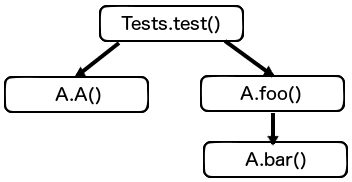
Figure 3.6: Representation of the *program slicing-based strategy (PS)*. Including: a) a block diagram with the main steps of the strategy; b) the commit blaming step, which matches changes slices with the covered statements and deduces responsible commits.

```
class A {
    public A(){ }
    public void foo() {  bar();  }
    public static void bar() { y = 17; }
    public static int y;
}


class Tests {
    public static void test() {
        A a = new A();
        a.foo();
    }
}
```

(a) Code snippet with changes highlighted by boxes.

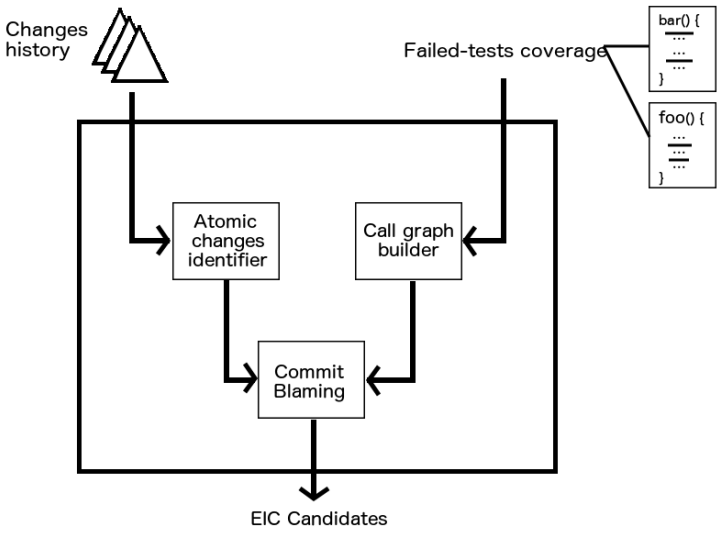


(b) Code changes represented as atomic changes.

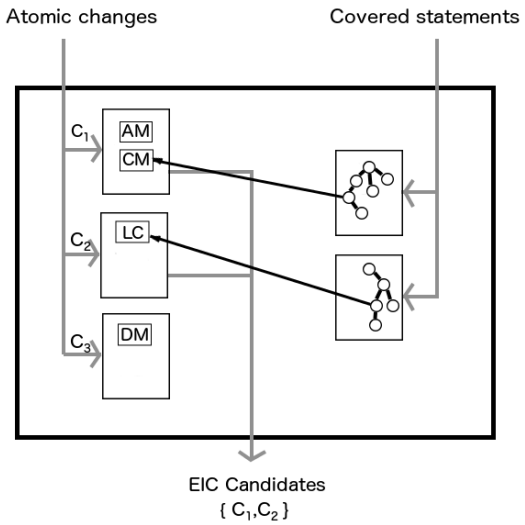


(c) Call graph built after the test behavior.

Figure 3.7: Example of change impact analysis based on atomic changes. Adapted from [RST+04]



(a) Block diagram.



(b) Commit blaming.

Figure 3.8: Representation of the *atomic changes-based strategy (ACS)*. Including: a) a block diagram with the main steps of the strategy; b) the commit blaming step, which matches statements covered by failed test to atomic changes that affect the test behavior.

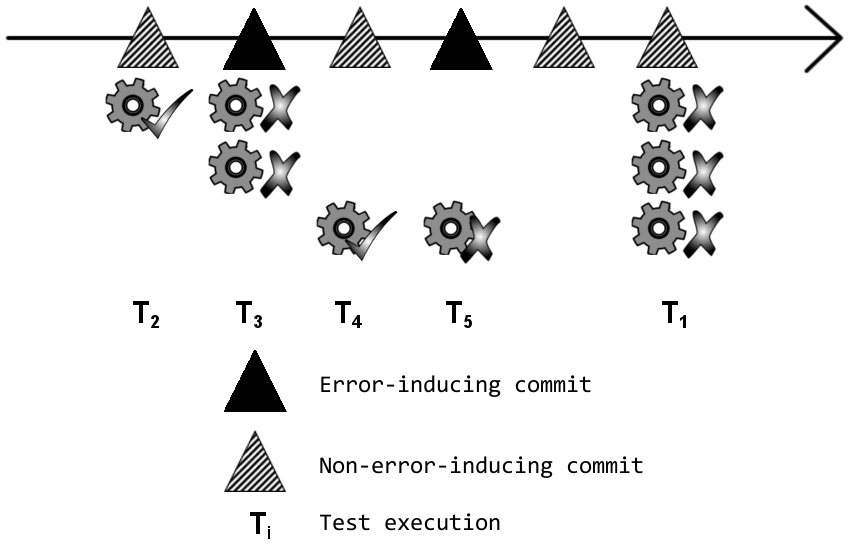


Figure 3.9: Concept of *rerun-all-failed-tests strategy (RAFT)*.

4 Reusing the Lazzer Framework

Contents

During the implementation phase of this thesis, the opportunity to reuse a framework for test prioritization and selection named *Lazzer* was identified. This software was developed by Christian Plewnia in context of his Master's Thesis[Ple15]. In the following sections a background of this framework and technologies used during the implementation phase, in addition to the extensions done to it, are described in detail.

4.1 Background

This section provides a background of the architecture constituting Lazzer, which ought to be helpful to understand the extensions done presented in the following section. Also, the different technologies used to implement the tracing strategies are introduced.

4.1.1 JUnit

JUnit is the first member of the testing frameworks family xUnit (see also section 3.1.3) to become widely known. As most testing frameworks that follow the xUnit pattern, JUnit provides its own format to specify test cases, assertions, set up phases, tear down phases, etc.

An example of a test class written with JUnit is presented in source code 4.1. Here can be identified the particular implementations of a typical xUnit pattern as follow:

- Test cases are written in the form of methods, each method is annotated with `@Test`. By using this label, the TestRunner can identify each of them.
- Assertions can be achieved by calling a set of functions: `assertTrue(...)`, `assertEquals(...)`, `assertNull(...)`, between others. Each of these function are accompanied with the expected and the real values to evaluate and, optionally, with a message which helps to localize a failed assertion.
- The instance variables inside this class represent the SUT. JUnit fully conforms to the xUnit pattern and provides the ability to define setup and teardown phases before and after each test method through the `@Before` and `@After` annotations. However, these phases can represent a significant overhead for a big collection of

```
1  public class JukeboxTest {
2      private static Jukebox jukebox;
3      private Song song;
4
5      @BeforeClass // Perform some setup before running any test method
6      public static void initOnce() {
7          jukebox = new Jukebox();
8      }
9
10     @Before // Perform some setup before each test method
11     public void initAlways() {
12         song = new Song("DJ Test", "My Unit");
13     }
14
15     @Test // this method is a test method
16     public void testAddSong() {
17         jukebox.addSong(song);                        // Execute
18         assertTrue(jukebox.existsSong("My Unit"));  // Verification
19     }
20 }
```

Source Code 4.1:
A Java code snippet showing an exemplary unit test written with JUnit. This test class includes a `Jukebox` object representing a SUT, which is initialized before any test is run inside a method annotated with `@BeforeClass`. Also, a `Song` object is initialized just before every test using the annotation `@Before`. Finally, the test method `testAddSong` (annotated with `Test`) verifies the correct behavior of adding a song to the jukebox, which is later confirmed using an `Assert` function.

tests, for this reason JUnit also provides the `@BeforeClass` and `@AfterClass` annotations, which implement the same functionality just before and after all test methods are run.

In its initial implementation, the Lazzer framework included a JUnit adapter as test class provider. Given the technology relevance, it represented a good opportunity to be reused within this thesis.

4.1.2 Git

Git is one of the most popular version control system used nowadays. It is built around the concept of *repository*, which is simply a database that stores all the necessary information to manage the versions and complete history of a project.

Differently to the traditional VCS, Git is distributed. In this sense, it does not copy the project from a central repository but *clones* the complete repository, and also uses its peers to pull new changes. Having a local repository allows a developer to work

autonomously and make as many commits to it as desired. A *branch* system allows, in addition, to have separate lines of development within the same repository.

Once a developer is ready to share its progress to the rest of the peers, each branch is paired with a *remote* and Git can be instructed to transfer the data to the remote following a push or pull model.

Internal structure

A Git repository is basically composed of two data structures [Loe09]: the *object store* and the *index*. The object store is the main component of the repository where all the original data files, log messages, author information, etc. are stored.

A repository is basically composed of four basic object types:

- *Blob*s represent a version of a file. Can be considered a raw piece of information without any metadata such as name or location.
- *Tree* objects represent one level of directory information. They hold enough information to identify the blobs in that directory and build a complete hierarchy of files and directories by referencing other trees.
- *Commit*s are in charge of holding the necessary information to represent a change introduced into the repository along with the author, date, log message, etc.
- *Tags*, whose function is to add a human-readable name to any object.

The *index* is a temporary binary file which captures the project structure at some moment in time. The key feature of an index is to provide the flexibility to specify changes of a version until they are ready to be *committed*.

Diff

A diff operation is an important action used within the tracing strategies. It involves performing a summary of differences between two items stored in the VCS. Diff allows to review a comparison between the current working directory and the staged content, between the current working directory and certain past commit (a common shortcut is to use the name *HEAD* to refer to the latest commit), or between two past commits.

Relevant to this work is the last operation, which outputs all modified files between the two trees accompanied with the modified sections between them. An example diff output is shown in figure 4.1.

4.1.3 Lazzer Testing Framework

As mentioned before, Lazzer was developed with the purpose of having a highly extensible, adaptable, technology-independent framework to evaluate test prioritization and selection strategies. With this idea in my mind, Lazzer provides the necessary hot-spots to add new stages to the test execution pipeline and also to integrate

```
Entry: DiffEntry[MODIFY TriangleTester/src/tester/TriangleTesterTopDown.java]
diff --git a/TriangleTester/src/tester/TriangleTesterTopDown.java b/TriangleTester/src/tester/TriangleTesterTopDown.java
index 4b8864c..47ed846 100755
--- a/TriangleTester/src/tester/TriangleTesterTopDown.java
+++ b/TriangleTester/src/tester/TriangleTesterTopDown.java
@@ -47,7 +47,7 @@
     }

     private boolean isEquilateral() {
-        return ((this.a == this.b) && (this.b == this.c));
+        return !((this.a == this.b) && (this.b == this.c)); // Negating this comparison will introduce an error
     }

     private boolean isScalene() {
```

Figure 4.1: Git diff command output.

technology-specific modules through provided hot-spots and their corresponding adapters. These modules can include test execution technologies (JUnit, NUnit, etc.), data store modules, as well as modules implementing different test prioritization and selection strategies.

Architecture

As mentioned before, the Lazzer architecture (cf. figure 4.2) consists of a set highly-decoupled modules, which jointly make up a framework capable of:

- Discovering Test classes in a project independently of the technology used to implement them. Currently, Lazzer includes a JUnit 4 Test Framework adapter which helps to identify the test classes following its specific formatting (see also section 4.1.1).
- Interfacing technology-specific data stores in order to gather pre-test execution information or storing post-test execution information regarding the execution of the tests.
- Selecting and prioritizing test cases based on specific strategies, which could require result information from previous test executions fetched previously from the data stores, or also be based only on the information gathered from the test classes, such as names, number of test methods, etc. The possibility of implementing new strategies is offered by the corresponding hot-spot and the framework launch configuration settings.
- Reporting test execution results using a *FreeMarker*-template-based adaptable format.

Reuse opportunity

The fact that Lazzer implements the whole lifecycle of tests, from their discovery, including execution, up to the delivery of results; also, considering that it provides the necessary hot spots to integrate new stages to the execution pipeline and to adapt

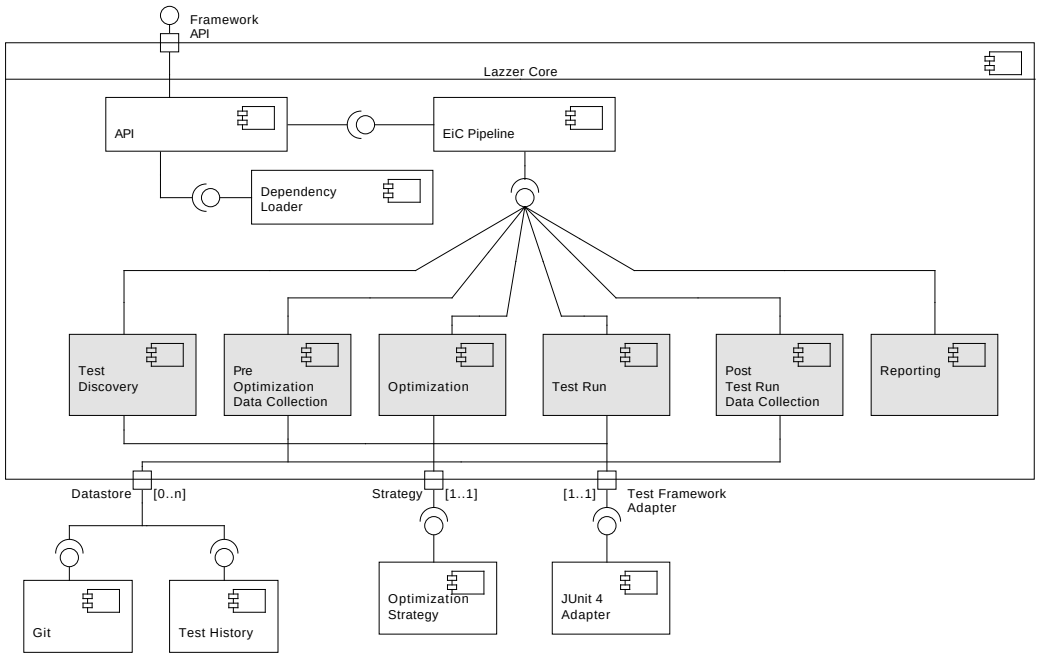


Figure 4.2: Original architecture of the Lazzer framework, including data stores (Git, Test history), Test optimization strategy, JUnit 4 Test Adapter.

different technologies, gives the opportunity to consider reusing the framework and adapting it for different purposes, such as the realization of this thesis.

The strategies conceived in the context of this thesis have as common starting point the output of the test execution, specifically the failed tests. In the presence of failed tests, the strategies focus on narrowing down the commits that could have possibly introduced an error in the project. Although other frameworks or plugins offer the possibility to run test classes and obtain a detailed report of the complying or failed tests (e.g., *Maven Surefire plugin*), one additional constrain is imposed by the *statement-test coverage-based strategy (STC)* (see section 3.4 and section 5), which requires obtaining fine-grained test coverage information, in order to match it with the commits that modified such code sections. Fortunately, Lazzer keeps track of the class files belonging to the project and coverage information can be collected by use of external libraries. Details regarding this will follow.

4.1.4 JaCoCo

JaCoCo is a code coverage library for Java that originally powered the Eclipse plugin EclEmma.[Ecl12] Nowadays, a number of tools rely on its capabilities in form of plugins for build automation tools (e.g., Gradle, Maven), CI tools (Jenkins, TeamCity), code inspection tools (SonarQube), etc.[Ecl09]

In the context of this thesis, JaCoCo was integrated into Lazzer in order to obtain test line-by-line coverage information and later use it to perform the matching between commit-change difference and the test line-coverage of the STC strategy (see also section 5).

Despite of the different tools available for Java code coverage, such as JCov[Ope15], Clover[Atl16], Cobertura[Cob16], CodeCover[Cod11], JaCoCo[Ecl16] resulted the most attractive tool based on the following factors:

Distribution scheme. JaCoCo is an open-source software, which means all of its features and source code are publicly available. Although Clover is full of features, the fact that it requires a license and its source code is not public, could have represented a project constrain.

Development activity. The fact that JaCoCo is actively supported increases the chance of continuing using the tool without being compromised by its lack of support of recent technologies, such as supporting the latest JDK (JaCoCo supports JDK 1.8 at the time of writing).

Coverage metrics. In contrast with some of the similar tools, JaCoCo provides up to line coverage, which is required for the STC strategy.

Build tools and CI servers integrations. JaCoCo offers provides integration to most popular of these tools, including (but not limited to) Maven, Gradle, Jenkins, TeamCity. Although at the end, it was opted not to use one of these specific adapters (see also section 4.2.2), its wide range of integration adapters imply a well-defined API and loose-coupled core, accompanied with a number of examples showing how to use the API reducing, with this, the learning curve.

This library relies on a program's byte code and *Intstrumentation* in order to be able to collect the coverage of a test execution. Code instrumentation is the process where a coverage tool introduces counters into code sections with the purpose of monitoring if a statement, function, branch, or condition was covered by a program execution.

Jacoco offers two modes to perform code instrumentation: *offline instrumentation*, where each class byte code is prepared before execution; *on-the-fly instrumentation*, process which skips the preliminary preparation and instruments the code as it is executed, this is achieved with the help of a *Java Agent* and represents less overhead.

Further details explaining how this library was integrated to the framework will be introduced in section 4.2.2.

4.2 Extensions to Lazzer

In this section, the extensions made to the Lazzer framework are detailed described.

4.2.1 Test result data store extension

The *test history data store* is one of the initial adapter example implementations of the Lazzer framework. Its main purpose was, as the name implies, to store test results and provide this information to test selection and sorting strategies which rely on previous test runs. This data store is realized using a Hibernate-based database whose schema is specified using Liquibase. Liquibase is a tool that allows tracking changes of database schema, which allows specifying schema changes in a number of formats, including XML, YAML, JSON and SQL).

The initial Lazzer implementation included the schema shown in figure 4.3a. However, the tracing techniques conceptualized in this work required relating past test runs to code versions, in order to make it possible to determine the last version whose tests ran successfully. The modified schema is shown in figure 4.3b. In addition to the schema, the corresponding *object-relational mapping (ORM)* was modified to adhere to the new schema.

4.2.2 Coverage gathering

Line-level-granular test coverage is one feature that powers the STC strategy (see also section 5) and it is obtained with the help of the Jacoco library. As mentioned before, this library can perform code instrumentation in two distinct modes: *offline* and *on-the-fly* instrumentation. For the implementation of the mentioned strategy it was opted to use the former approach, since the latter requires the Java Agent to be attached to a process (Test executor) whenever it is launched in a *Java virtual machine (JVM)*, and in Lazzer the tests are executed internally, without launching a separate process.

In this regard, the offline instrumentation was introduced as a separate stage in the Lazzer's core pipeline. An additional step involves stopping the *coverage runtime*. This step was also integrated in the Lazzer's core pipeline and take place just before and after test execution. This stage, together with the preparation (instrumentation) stage are depicted in figure 4.4.

4.2.3 Tracing stage hot-spot

One of the main objectives of implementing the tracing strategies conceived in this work was to be able to execute and compare results of each of them. With this idea in mind, and taking advantage of Lazzer's architecture extensibility, a new hot-spot was implemented and incorporated into Lazzer's execution flow as a separate pipeline stage. This modification is represented also in figure 4.4.

4.2.4 Tracing reporting

The current implementation of the tracing strategies leverages the flexibility of Lazzer in terms of customizing the output report. The reporting step in Lazzer takes place in a separate stage (see figure 4.2) and makes it possible to specify the console output format using a template engine named *FreeMarker*. Such template engine allows to specify a

base output format in a file, which can be processed inside the program and combined with the application-specific data model to produce the formatted output. The engine is powerful enough to manipulate data collections and different data types required to create a rich output.

As shown in figure 4.5, the result of the tracing stage represents a collection of `TracingOutput` objects which is passed over to the reporting stage. Each of these objects contains the results of the tracing strategies executed by the framework.

The information contained within these objects include:

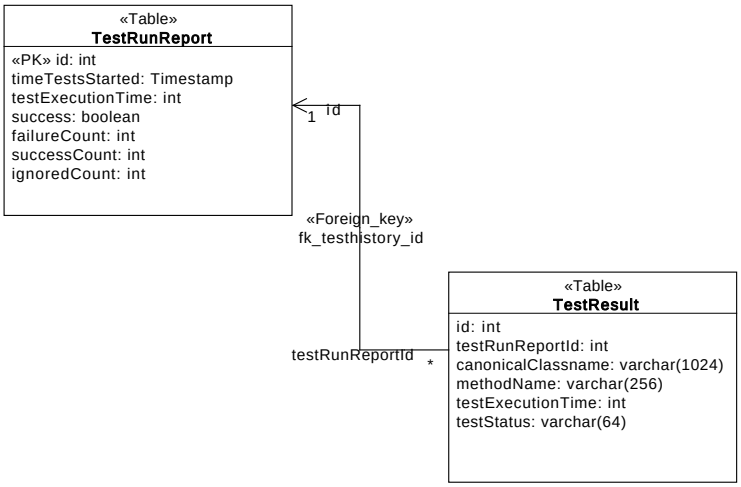
StrategyName. The strategy whose this results belong to.

LastSuccesfulCommit. The past commit from which the strategy started the analysis, i.e., the last codebase version that logged a complete successful test execution.

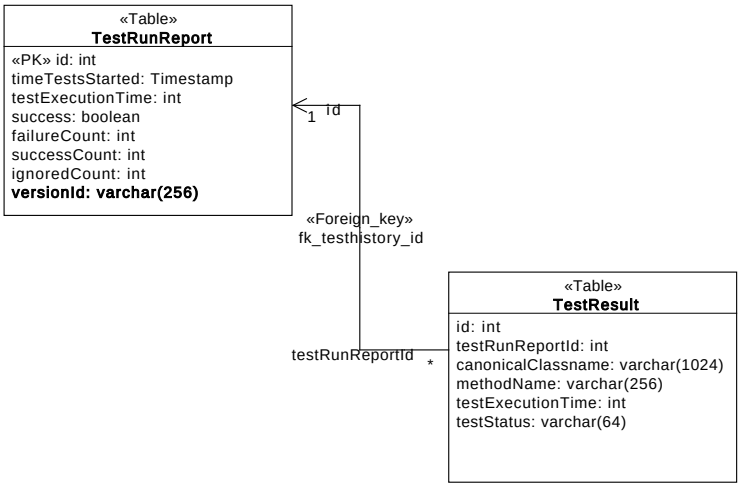
CurrentVersionId. The codebase version that was tested.

TracingResult. The `Map` structure relating each of the failed tests to the set of EIC that may be have introduced an error.

The result of the transformation performed by the template engine using the adapted format can be observed in figure 4.6.



(a) Original schema.



(b) Modified schema.

Figure 4.3: Test history data store schema. Addition of the *versionId* field helps to track the source code version being tested.

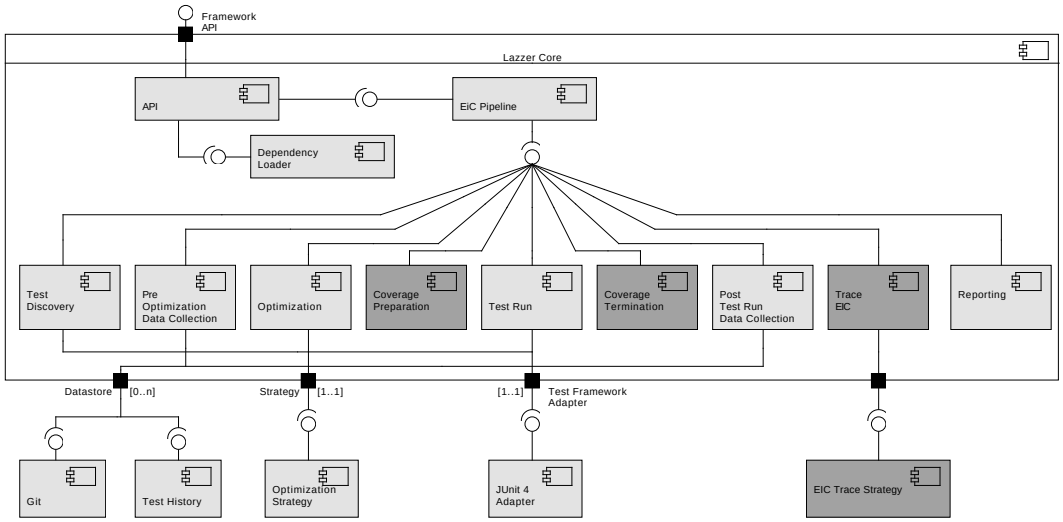


Figure 4.4: Modifications (dark gray) done to Lazzer's core, including: coverage preparation (instrumentation) stage, coverage termination stage, and tracing strategy hot spot.

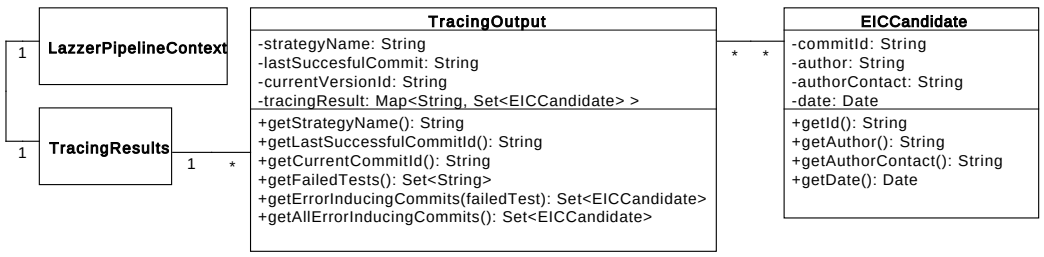


Figure 4.5: Tracing result class diagram.

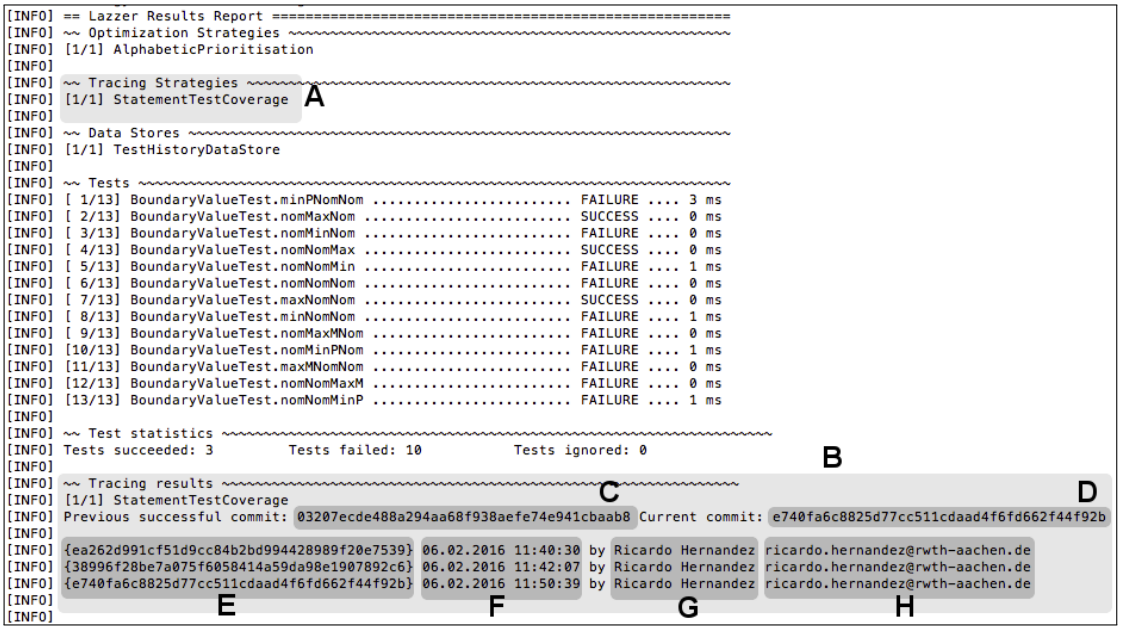


Figure 4.6: Tracing console output displaying: (a) a listing of the tracing strategies executed, (b) a comprehensive report for each of the strategies, (c) the last successfully tested codebase version, (d) the current codebase version being tested, (e) each of the EIC ids, (f) date and time of the commit, (g) author of the commit, and (h) contact of the commit author.

5 Statement-test coverage strategy realization

Contents

After addressing the major modifications done to the Lazzer framework to enable EIC tracing in the last section, it is time to review the enhancements done to it that are specific to the *statement-test coverage-based strategy (STC)*. This enhancements comprise the information retrieval regarding the past commit changes and the coverage-changes matching, described next.

5.1 Past changes information retrieval

The past changes information retrieval consists in a diff operation that takes place inside Lazzer's Git (see section 4.1.2 and section 4.1.3) module, one of the data stores implementation that is concerned with the codebase storage in a repository. This diff operation is carried out for each code version that is analyzed starting from the last record of a STCV up to the most recent version. The realization of this strategy implemented in Lazzer includes the considerations mentioned in section 3.4 including the support of *DeltaSpots*.

Internally, the Git data store module relies on three components (Figure 5.1) to process the diff information: an implementation of `OutputStream`, that helps to gather the raw output from the diff operation (`DiffFormatter`); a `DiffOutputAnalyzer`, which parses the output considering the diff format and extracts the modified source lines; `EffectiveLineUpdater`, an object that keeps track of the DeltaSpots and updates the modified source lines of each file version accordingly.

The class `DiffOutputAnalyzer` creates objects of type `CommitDiffResult`, which contains the modified lines for certain file in a code version. These are collected by `HistoryDiffData`, the output data structure.

The output of this step is a data structure that relates the project's modified files that were detected to the responsible commit, including the diff information with the modified source lines. This structure is represented in figure 5.2.

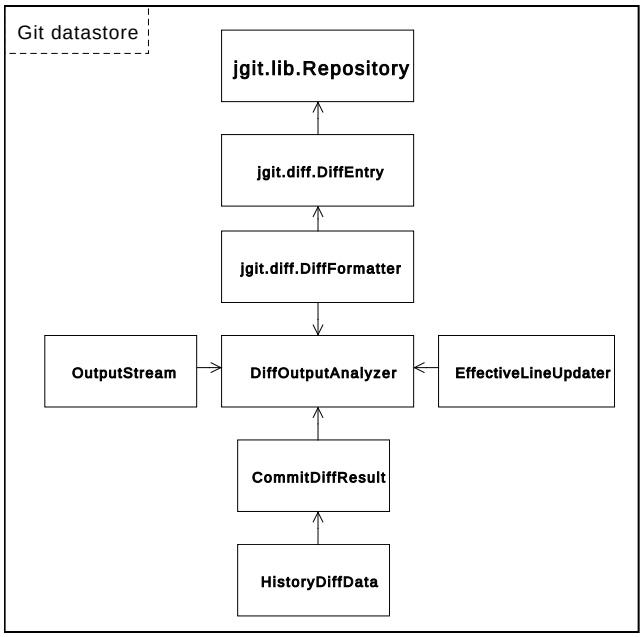


Figure 5.1: Class relationship diagram inside Git datastore.

5.2 Changes-coverage matching

Another key step concerning the strategy involves the matching between the collected modified lines and the covered lines. In order to do this, both data should be available to the strategy inside the strategy context.

The operation of matching the data mentioned before takes place in a support class named `ImpactAnalyzer`, which basically goes through the covered classes of each failed test and queries the changes information data for records of the class. If found, and after performing a line intersection comparison, the commit information related to the change is collected as a EIC-candidate. After consulting all failed tests, the set of EIC-candidates is delivered to the output. It is worth mentioning that, due to the internal implementation of the `EICCandidate` object, object duplicates are avoided.

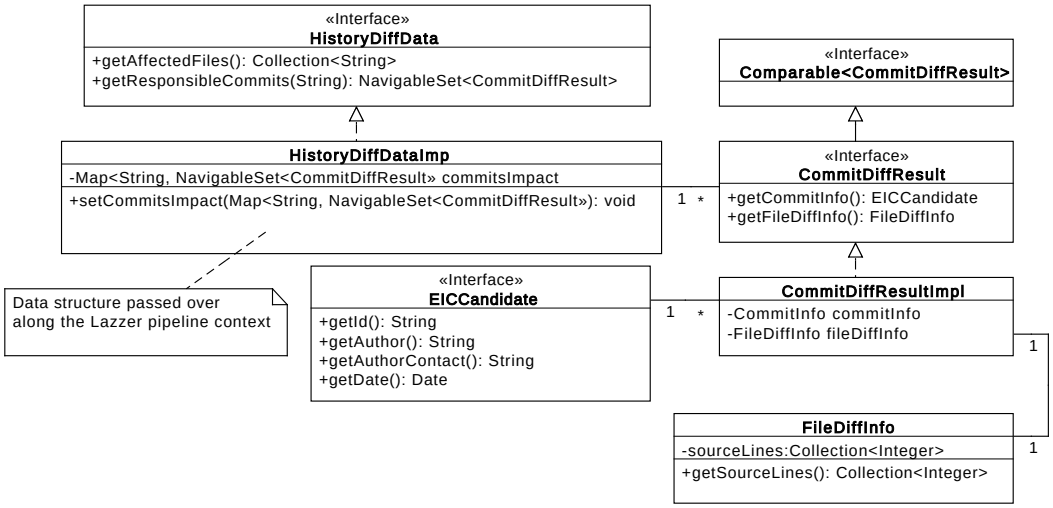


Figure 5.2: Class diagram representing the output data structure of the change information retrieval step.

6 Evaluation

Contents

Having reviewed the concept of the tracing strategies and their implementation details, it is now turn to evaluate them. For this purpose a set of demo scenarios were considered, as described next.

6.1 Demo scenarios

In this section, we present a collection of cases which represent development scenarios where errors could be inadvertently introduced into a codebase while performing common tasks, such as: maintenance, feature development, or fixing other bugs.

These scenarios aim to demonstrate the behavior of the implemented strategy under such conditions.

6.1.1 Demo project

The project used to demonstrate the functionality of the implemented strategy consists in a implementation of a popular Triangle type evaluator which, based on the side length parameters received as input, can determine either the type of triangle in concern: *Equilateral*, *Isosceles*, *Scalene*; or confirm an invalid sides configuration for a triangle.

A class diagram of this project is depicted in figure 6.1 showing `TriangleTester`, the *system under test (SUT)* and the test suite `TriangleTesterTopDown` which evaluates the output of the SUT after providing it with different input configurations (the full set of test methods has been omitted in the test suite for simplicity).

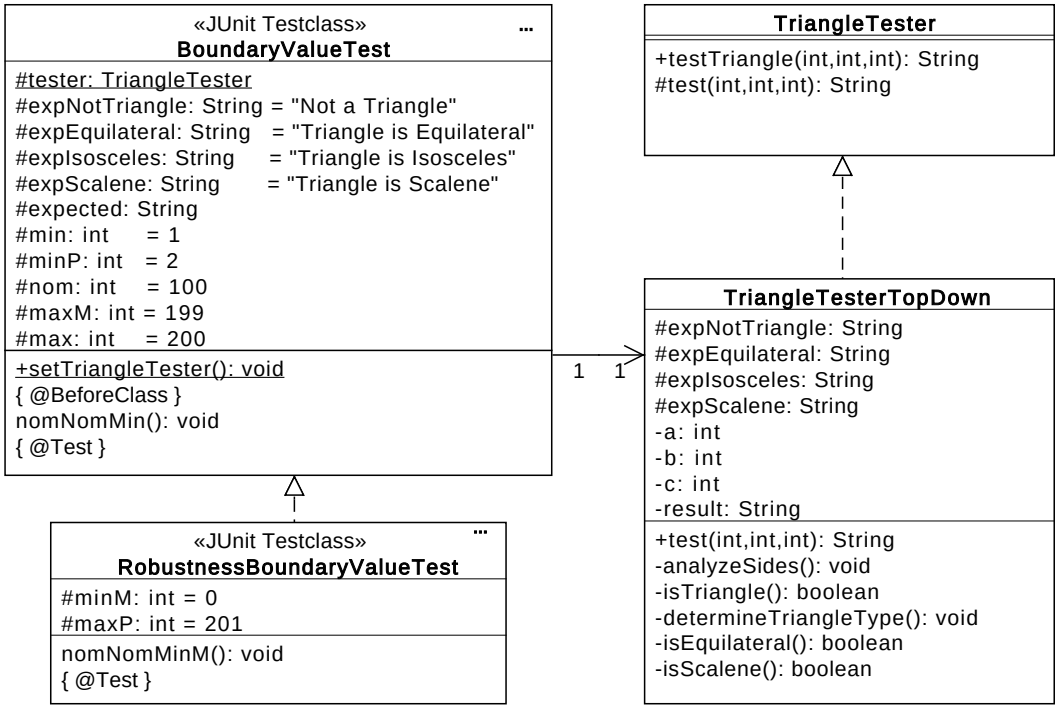


Figure 6.1: Class diagram depicting the relation of the class.

6.1.2 Simple line modification

The aim of this scenario (Figure 6.2) is to show the functionality of the strategy as it detects a simple error introduced in a code line in the class `TriangleTesterTopDown` (Figure 6.2b). This modification impacts the result expected by the test suite yielding a failed test execution.

The strategy focuses on localizing the EIC despite of existing a couple of later commits which also influenced the codebase (Figure 6.2a). The results of the execution can be observed in figure 6.2c. This figure confirms that the EIC in question was indeed found, however, the two later commits were also within the candidates, since they recorded changes in sections covered by the failing tests.

6.1.3 Displaced error line

The second demo scenario (Figure 6.3) presented here consists again in an induced error in the code of class `TriangleTesterTopDown`, however, an additional constraint is represented by a later commit (see figure 6.3a) introducing a set of empty lines that displace the original *faulty* line and impose an additional challenge when matching the modified/covered lines (Figure 6.3b).

As it can be appreciated in the results (Figure 6.3c), the EIC was identified together with two later commits that modified test-covered sections, hence, the implementation of the strategy is able to track the changes of source code lines numbering, in order to be able to match the corresponding modified/covered lines.

6.1.4 Refactoring: Method extraction

Refactoring is a common operation performed during the software development lifecycle. Method extraction is a well-known refactoring pattern. We will use such pattern in another scenario to demonstrate the functionality of the strategy.

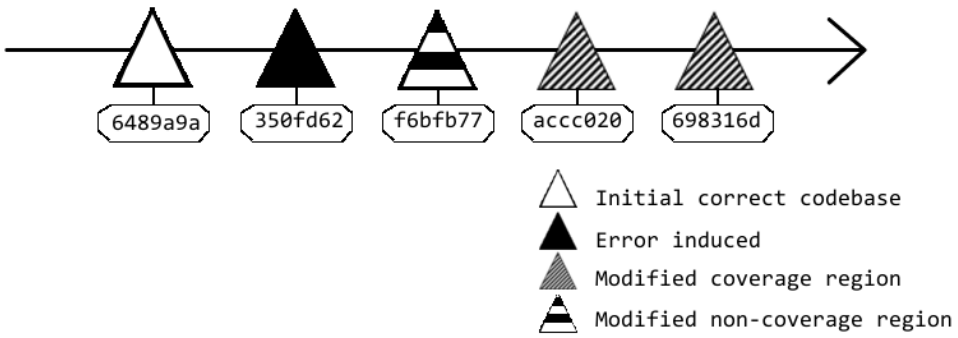
The scenario in question is shown in figure 6.4. This scenario consists in, similarly to the previous scenarios, a commit (Figure 6.5a) introducing an error where, in this case, involves an interchanged assignment statements. This commit is followed by the refactoring operation which modifies the location of the changed code lines (Figure 6.4b) and represents a challenge for the strategy. The result of the strategy is shown in figure 6.5d, which includes only the commit that performed the refactoring and modified a code section covered by the tests. Unfortunately, this operation hindered the discovery of the real EIC.

6.1.5 Refactoring: Pull-up

Our last demo scenario (Figure 6.5) involves another common refactoring operation: the class fields pull-up pattern. This pattern involves moving class fields from a subclass to any of its parent classes in order to generalize its definition.

After introducing an error in a commit by modifying the constant values of the fields (Figure 6.5a), the refactoring operation (Figure 6.5b and Figure 6.5c) challenges the strategy to discover it. The results of this effort are shown in figure 6.5d, which shows only the last commit whose changes modified a test-covered section.

At this point it is worth mentioning that the changes done to the class fields is not considered a test-covered section, this is due to the impossibility of the coverage gathering framework to detect coverage of class fields, given that the framework relies on byte-code and these elements are not translated to byte-code during build time.



(a) Commit history timeline.

```
    }
}

private boolean isEquilateral() {
    return ((this.a == this.b) && (this.b == this.c));
}

private boolean isScalene() {
    return ((this.a != this.b) && (this.a != this.c) &&
}
```

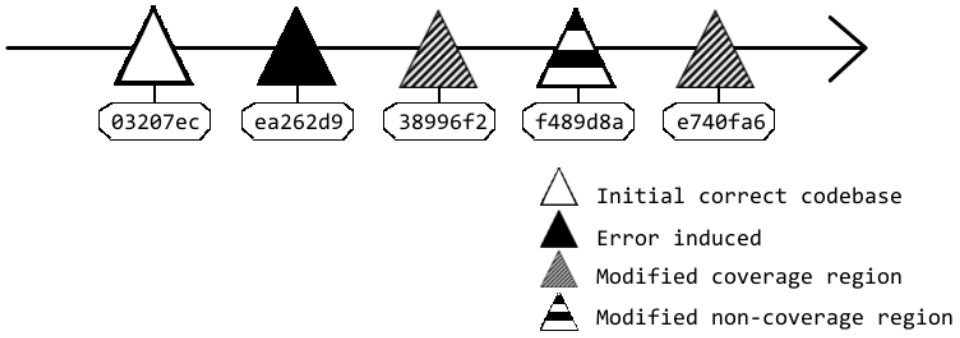
```
    }
}

private boolean isEquilateral() {
    return !((this.a == this.b) && (this.b == this.c)); // Negating this comparison will i
}

private boolean isScalene() {
    return ((this.a != this.b) && (this.a != this.c) && (this.b != this.c));
}
```

(b) Diff view of the modification done to a single code line.

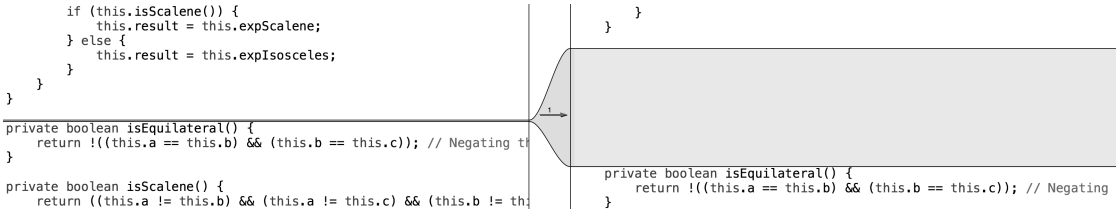
```
~~ Tracing results ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[1/1] StatementTestCoverage
Previous successful commit: 6489a9aacdbfe002323a5f797cad701609b1ceff Current commit: 698316d24abdb3b8e9513b29f9e2dc556929535c

{350fd6280ca56b505169f86ea99619cd9f4540df} 04.02.2016 16:34:42 by Ricardo Hernandez ricardo.hernandez@rwth-aachen.de
{accc020a7b1cc80006f5ef04d1b2fe9526b52695} 04.02.2016 16:41:38 by Ricardo Hernandez ricardo.hernandez@rwth-aachen.de
{698316d24abdb3b8e9513b29f9e2dc556929535c} 04.02.2016 16:44:25 by Ricardo Hernandez ricardo.hernandez@rwth-aachen.de
```

(c) The output of the strategy.

Figure 6.2: Demo scenario consisting in an error induced by a simple line modification in the codebase. Figure shows: a) the scenario's commit history with an EIC and later commits affecting covered and non-covered sections, b) change done to class `TriangleTesterTopDown` in commit `350fd62` to induce the error, and c) the output of the strategy showing the identification of the real EIC with two later non-valid EICs.



(a) Commit history timeline.

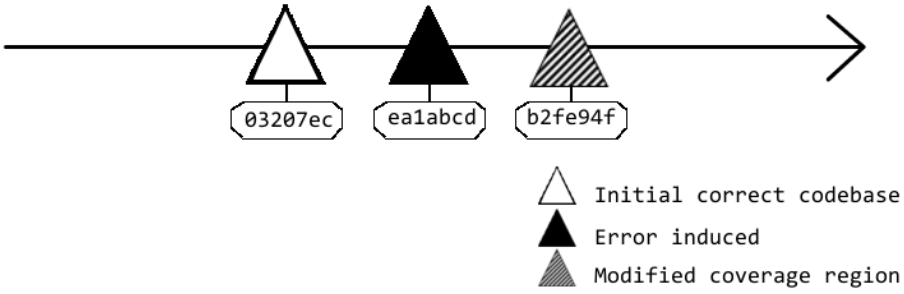


(b) Diff view of a displaced *faulty* code line in `TriangleTesterTopDown`.

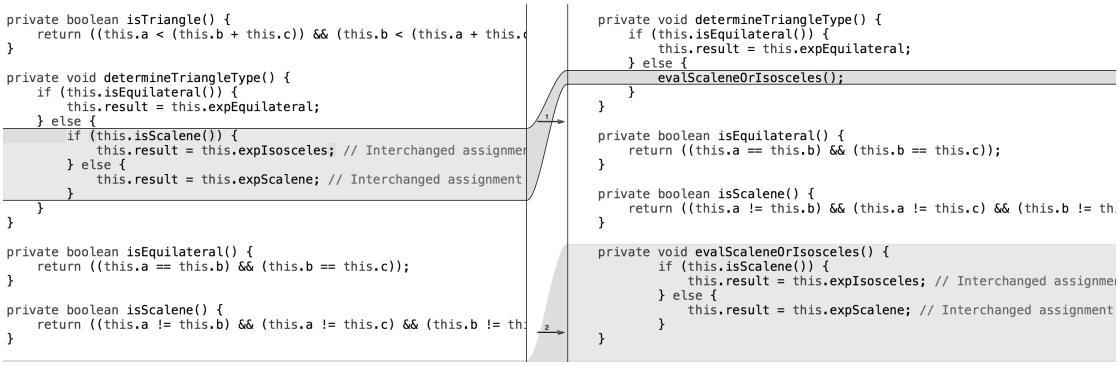
```
~~ Tracing results ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[1/1] StatementTestCoverage
Previous successful commit: 03207ecde488a294aa68f938aefe74e941cbaab8 Current commit: e740fa6c8825d77cc511cdaad4f6fd662f44f92b

{ea262d991cf51d9cc84b2bd994428989f20e7539} 06.02.2016 11:40:30 by Ricardo Hernandez ricardo.hernandez@rwth-aachen.de
{38996f28be7a075f6058414a59da98e1907892c6} 06.02.2016 11:42:07 by Ricardo Hernandez ricardo.hernandez@rwth-aachen.de
{e740fa6c8825d77cc511cdaad4f6fd662f44f92b} 06.02.2016 11:50:39 by Ricardo Hernandez ricardo.hernandez@rwth-aachen.de
```

(c) The output of the strategy.

Figure 6.3: Demo scenario consisting in an error induced by a line modification which is displaced in a later commit. Figure shows: a) the scenario's commit history with an EIC and a later commit modifying the location of the faulty line, b) change done to class `TriangleTesterTopDown` in commit `f489d8a` to displace the faulty line, and c) the output of the strategy showing the identification of the real EIC with two later non-valid EICs.



(a) Commit history timeline.

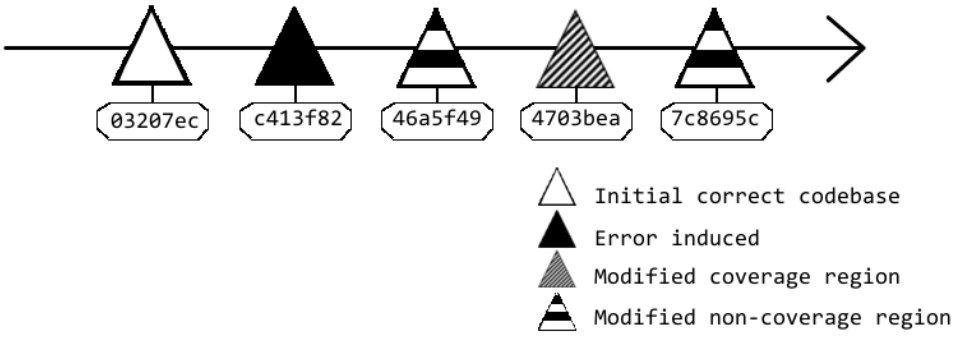


(b) Diff view of a refactoring operation consisting in an extraction of a code section into a separate method.

```
~~ Tracing results ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[1/1] StatementTestCoverage
Previous successful commit: 03207ecde488a294aa68f938aefe74e941cbaab8 Current commit: b2fe94f73a01c1922c3030ecb89782f3c671a443

{b2fe94f73a01c1922c3030ecb89782f3c671a443} 01.03.2016 23:54:08 by Ricardo Hernandez ricardo.hernandez@rwth-aachen.de
```

(c) The output of the strategy.

Figure 6.4: Demo scenario consisting in an error hidden by a method extraction refactoring operation. Figure shows: a) the scenario's commit history with an EIC and a later commit where the refactoring operation was performed, b) refactoring done to class `TriangleTesterTopDown` in commit `b2fe94f`, and c) the output of the strategy showing the identification of only one commit that modified a test-covered section.



(a) Commit history timeline.

```
ic class TriangleTesterTopDown extends TriangleTester {

protected final String expNotTriangle = "Not a Triangle";
protected final String expEquilateral = "Triangle is Equilateral!!!";
protected final String expIsosceles = "Triangle is Isosceles";
protected final String expScalene = "Triangle is Scalene";

private int a;
private int b;
private int c;
private String result;

@Override
public String test(final int aSeite, final int bSeite, final int cSeit
    this.a = aSeite;
    this.b = bSeite;
```

```
public class TriangleTesterTopDown extends TriangleTester {

    private int a;
    private int b;
    private int c;
    private String result;

    @Override
    public String test(final int aSeite, final int bSeite,
        this.a = aSeite;
        this.b = bSeite;
        this.c = cSeite;
        this.analyzeSides();
        return this.result;

    }
```

(b) Pull-up refactoring operation in class `TriangleTesterTopDown`.

```
ic abstract class TriangleTester {

public String testTriangle(final int a, final int b, final int c)
        throws OutOfBoundsException {
    final int min = 1;
    final int max = 200;
    if ((a < min) || (b < min) || (c < min)) {
        throw new OutOfBoundsException();
    }
    if ((a > max) || (b > max) || (c > max)) {
        throw new OutOfBoundsException();
    }

    return this.test(a, b, c);
}

protected abstract String test(int a, int b, int c);
```

```
public abstract class TriangleTester {

    protected final String expNotTriangle = "Not a Triangle";
    protected final String expEquilateral = "Triangle is Equilateral!!!";
    protected final String expIsosceles = "Triangle is Isosceles";
    protected final String expScalene = "Triangle is Scalene";

    public String testTriangle(final int a, final int b, final int c)
            throws OutOfBoundsException {
        final int min = 1;
        final int max = 200;
        if ((a < min) || (b < min) || (c < min)) {
            throw new OutOfBoundsException();
        }
        if ((a > max) || (b > max) || (c > max)) {
            throw new OutOfBoundsException();
        }
```

(c) Pull-up refactoring operation in class `TriangleTester`.

```
~~ Tracing results ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[1/1] StatementTestCoverage
Previous successful commit: 03207ecde488a294aa68f938aefe74e941cbaab8 Current commit: 7c8695c4949513d19e74fe7ad15031fd413a5bd6

{4703beab22b41bda15e18c4c45df28469b638cf2} 24.02.2016 09:59:54 by Ricardo Hernandez ricardo.hernandez@rwth-aachen.de
```

(d) The output of the strategy.

Figure 6.5: Demo scenario consisting in an error hidden by a pull-up refactoring operation. Figure shows: a) the scenario's commit history with an EIC and a later commit where the refactoring operation was performed, b) and c) refactoring done between classes `TriangleTesterTopDown` and `TriangleTester` in commit `46a5f49`, and c) the output of the strategy showing the identification of only one commit that modified a test-covered section.

7 Discussion

Contents

Having introduced each of the strategies concepts, the schema of properties, and after demonstrating the realization of the STC strategy in common development scenarios, it is turn of discussing the behavior observed and categorize the strategies according to the schema.

7.1 Strategies properties

Previously, a properties schema (see section 3.3.2) was introduced that can be used to catalogue tracing strategies. Now, we will focus on matching each of the properties to the strategies conceptualized in this work.

The first strategy to address is the *statement-test coverage-based strategy (STC)*. As seen before, this strategy is based on source code textual comparisons, hence no syntactic nor semantic information is used during the EIC localization. This might impact the results delivered, meaning that any matching modified/covered source lines will result in the commit under evaluation to be considered an EIC candidate. For this reason, the presence of false-positives could be expected, hence this strategy is not precise. Also, code changes history may result in code sections being deleted, information about the affected lines can be lost at a later comparison, resulting in possible EIC being omitted. Therefore, this strategy cannot be considered safe either. Regarding reliability, the technique does not enforce a systematic approach to deliver at least one valid EIC, hence it is not reliable.

The *program slicing-based strategy (PS)* is the second strategy conceived. Taking advantage of static program slicing provides syntactic information that allows to identify the impact of each commit change. Given that it is a static approach, it is restricted to provide the *probably* affected execution paths (while exact execution paths can only be determined with a defined input), hence there is the possibility that a commit is considered responsible even though the error might be introduced in a different commit, leading to a false-positive result, which yields to a non-precise strategy. In regards to the safety property, we should recall that the strategy relies on syntactic information. In this sense, code changes that represent semantic modifications (i.e., dynamic dispatching) may be ignored, due to this reason it cannot be considered safe. Similarly to the previous

technique, it does not enforce a systematic approach to deliver at least one valid EIC, hence it is not reliable either.

The third strategy to consider is the *atomic changes-based strategy (ACS)*. The strategy is based on the change impact analysis by Ren, et al. which is able to safely detect regression tests whose behavior has been affected by code changes.[RST+04] Since our work builds upon that technique and our strategy relies on failed test results to detect EIC, we can therefore conclude this strategy is safe. In regards to precision, contrary to the previous technique, the method call relations can be gathered dynamically during test execution. In this sense, the probably-affected regions are gathered more accurately. However, this technique is more coarse-grained and it does not consider statement-level dependencies. Therefore, it can not be considered a precise technique. Similarly to the two previous techniques, a lack of enforcement to deliver at least one valid EIC, results in a non-reliable technique.

Finally, the *rerun-all-failed-tests strategy (RAFT)* focuses on delivering a valid EIC at the minimum within the results. Although the followed approach might represent an expensive computation overhead, it tests thoroughly all code versions until one execution fails. Moreover, it will not stop when it finds the first failing code version, instead it will continue evaluating the following versions until all test cases have evidently failed and a *responsible* commit has been found for each of them. By following this approach, the strategy will clearly return at least one EIC candidate, hence it is reliable. In addition, it collects all code version with a failing test execution, therefore it is safe, as well. However, since an error in certain code version propagates in the following versions, there exist the possibility of false-positives, consequently it is not a precise strategy.

Having analyzed all the strategies conceived in this work, their corresponding properties according to the schema presented here can be summarized in table 7.1.

Strategy	**Safe**	**Precise**	**Reliable**
STC	-	-	-
PS	-	-	-
ACS	X	-	X
RAFT	X	-	X

Table 7.1: The properties associated to each of the tracing strategies following the schema presented in this work.

7.2 Strengths and weaknesses

In the last section, the approaches of each strategy were analyzed to determine the categories that are associated with them. Within that analysis, reasonings were provided concerning why they could include false-positives and false-negatives within their resulting candidates.

Table 7.2 summarizes the arguments form last section regarding the strength or weakness of finding valid EICs. Computation overhead is not the main criteria of this comparison, however an argument may be included if it represents a significant difference against the rest of the strategies.

Strategy	**Strengths**	**Weaknesses**
STC	- Diff information available after extracting changes history, no additional computation.	- No semantic information due to restriction to textual comparison. - Difficult to recover information from deleted lines.
PS	- Syntactic information yields possible execution paths.	- Lack of semantic information.
ACS	- Atomic changes translate code changes into impact-relevant semantic data.	- Coarse-grained precision based on method-level call relationships.
RAFT	- Determines the valid EIC(s) after an extensive evaluation.	- Expensive strategy that potentially requires evaluating each commit.

Table 7.2: Comparison of the strengths and weaknesses of the strategies.

8 Conclusion

Contents

The final chapter of this thesis includes a brief recapitulation of the contributions made in this work (Section 8.1). Moreover, section 8.2 provides an outlook on future work directions regarding the conception and realization of new tracing strategies, as well as, improvement on the ones included in this work.

8.1 Summary

This work introduced the concept of *error-inducing commits (EICs)*, as code commits made to a codebase whose failed test execution has evidenced the introduction of an error. Also, it conceptualized strategies to localize them in a CI environment.

Chapter 2 started off with an overview of the techniques that allow identifying the impact of changes in the area of *regression testing*. It reviewed, as well, existing approaches that assist on validating the code contributions of developers before integrating them into the project codebase.

Later, chapter 3 elaborated on the concept of EIC and the goal of the tracing strategies. Furthermore, a set of strategies that build upon the techniques reviewed in chapter 2 were introduced. Additionally, a property schema used to categorize strategies was provided.

Chapter 4 went beyond the concept and reused the framework for regression test selection and prioritization to realize our strategy concepts. Specific details regarding the implementation of one strategy were covered in chapter 5.

In chapter 6 the behavior of the implemented strategy was shown under common software development scenarios, this helped to understand the limitations of the strategy. This limitations, together with the properties according to the schema provided in this work, were addressed in chapter 7 with an extensive discussion.

8.2 Future Work

This thesis provided the grounds in the field related to discovering EIC in a CI environment. As this is a research line that can be explored further, we will point out to suggestions that can be considered regarding the strategies included in this work or potential ideas for new strategies.

8.2.1 Strategies realization

This thesis included only one implementation of the conceptualized tracing strategies. Suggestions regarding the implementation of the rest of the strategies as well as improvements to the current one will be done next.

STC strategy realization improvement

The evaluation of the *statement-test coverage-based strategy (STC)* conducted in section 6.1 based on common development scenarios evidenced the capabilities and the limitations of the strategy. As it was discussed before, this strategy is based on source code textual comparisons, therefore it is challenging to extract semantic information from it. However, this approach can still be improved to reduce the amount of false-positives. An improvement to this approach can be found in the work of Kim et al. [KZPJ06], where *annotation graphs* are employed to keep track of added, deleted or modified lines across versions in a graph-based structure, which allows to compute backward deep-first searches in order to identify the source code dependencies. An even further improvement was implemented by Williams and Spacco [WS08] who, instead of relying on the code changes information provided by the CVS *annotation* command, track individual code line numbers and are able to track line dependencies even across larger code changes.

Further strategies realizations

Due to time constraints, this thesis included only the implementation of the STC strategy. However, it would be relevant to realize the other strategies introduced in this work. Although it would be necessary to rely on additional software or libraries, the Lazzer framework and the modifications done in this work provide the basis for realizing them in the future. It is worth pointing out a few considerations regarding the implementation of specific strategies:

PS strategy. The program slicing-based strategy relies on the analysis of code dependencies using method call graphs and program slicing. There are several tools that can enable this. However, two of them that are worth considering using could be WALA[IBM15] and Chord[Nai], this is based on the evidence of continuing being supported and the amount and quality of their documentation, which might facilitate the integration for tracing strategies.

ACS strategy. Change impact analysis based on atomic changes was the contribution of the work by Ren et al. [RST+04]. Their work was realized in form of a plug-in for Eclipse which could aid the tasks of developers. Although this plug-in does not seem to be available anymore [Wlo08], posterior work has made use of the tool, which could point to the location of the original implementation[SRRT06].

8.2.2 Strategy conception

This work considered code analysis techniques that have their origins in CIA in order to conceive tracing strategies. These techniques, however, are not the only ones existing. Symbolic execution analysis[Leh15] is another technique that could be explored in a future to trace EICs.

8.2.3 Real world evaluation

Although this work performed experiments using only a small project, efforts were done to evaluate the implemented strategy in a large and real-world project. Given that many open-source programs make their source code widely available through repositories and access to their continuous integration monitoring is also available, experimenting with them should not be feasible. A difficulty was found, however, when looking for evidences of broken builds due to failed tests. A reason could lie in the fact that developers perform testing before committing their work back to the project repository. Therefore, a more sophisticated experiment configuration may be required.

Bibliography

[ACD07] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. Learning from bug-introducing changes to prevent fault prone code. *Ninth international workshop on Principles of software evolution in conjunction with the 6th ESECFSE joint meeting IWPSE 07*, page 19, 2007. URL: http://portal.acm.org/citation.cfm?id=1294948.1294954, doi:10.1145/1294948.1294954.

[Apa15] Apache. Subversion, 2015. URL: https://subversion.apache.org.

[Atl16] Atlassian. Clover, 2016. URL: https://www.atlassian.com/software/clover.

[Bec02] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[BRO13] Marcel Böhme, Abhik Roychoudhury, and Bruno C d S Oliveira. Regression Testing of Evolving Programs. *Advances in Computers*, 89:53–88, 2013. doi:10.1016/B978-0-12-408094-2.00002-3.

[Cob16] Cobertura. A code coverage utility for Java, mar 2016. URL: http://cobertura.github.io/cobertura/.

[Cod11] CodeCover. Code coverage tool for Java, jul 2011. URL: http://codecover.org/.

[DMG07] Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, first edition, 2007.

[Ecl09] EclEmma. JaCoCo tool integrations, 2009. URL: http://eclemma.org/jacoco/trunk/doc/integrations.html.

[Ecl12] EclEmma. Java Code Coverage for Eclipse, sep 2012. URL: http://eclemma.org/.

[Ecl16] EclEmma. JaCoCo Java Code Coverage Library, feb 2016. URL: http://eclemma.org/jacoco/.

[Fac15] Facebook. Big Code: Developer Infrastructure at Facebook's Scale, mar 2015. URL: https://developers.facebooklive.com/videos/561.

[FFB13] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, 2013. doi:10.1109/MIC.2013.25.

[Fow06] Martin Fowler. Continuous Integration, may 2006. URL: http://martinfowler.com/articles/continuousIntegration.html.

[Fow11] Martin Fowler. Xunit. *Martin Fowler. Available at: http://www.martinfowler.com/bliki/Xunit. html*, 2011.

[GHK+01] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porters, and Gregg Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001. URL: http://portal.acm.org/citation.cfm?doid=367008.367020, doi:10.1145/367008.367020.

[Git12] Git. Git bisect, dec 2012. URL: https://git-scm.com/docs/git-bisect-lk2009.html.

[Git16] Git. Git distributed SCM, mar 2016. URL: https://git-scm.com/.

[IBM15] IBM T.J. Watson Research Center. T.J. Watson Libraries for Analysis (WALA), 2015. URL: http://wala.sourceforge.net/.

[Kum10] Ashish Kumar. Development at the speed and scale of google, 2010. URL: http://www.infoq.com/presentations/Development-at-Google.

[KZPJ06] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead. Automatic identification of bug-introducing changes. In *Proceedings - 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006*, pages 81–90, 2006. doi:10.1109/ASE.2006.23.

[Leh11a] Steffen Lehnert. A Review of Software Change Impact Analysis. *Technology*, page 39, 2011. URL: http://www.db-thueringen.de/servlets/DocumentServlet?id=19544.

[Leh11b] Steffen Lehnert. A taxonomy for software change impact analysis. *Proceedings of the 12th international workshop and the 7th annual ERCIM workshop on Principles on software evolution and software evolution - IWPSE-EVOL '11*, (1):41, 2011. URL: http://dl.acm.org/citation.cfm?doid=2024445.2024454, doi:10.1145/2024445.2024454.

[Leh15] Steffen Lehnert. *Multiperspective Change Impact Analysis to Support Software Maintenance and Reengineering*. PhD thesis, 2015.

[Lig09] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. 2009. doi:10.1007/978-3-540-76323-9.

[Loe09] Jon Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, Inc., 1st edition, 2009.

[Mer] Mercurial. Bisect. URL: `https://selenic.com/hg/help/bisect`.

[Mer16] Mercurial. Mercurial distributed SCM, mar 2016. URL: `https://www.mercurial-scm.org/`.

[Mes06] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[Nai] Mayur Naik. Chord: A Program Analysis Platform for Java. URL: `http://www.cc.gatech.edu/{~}naik/chord.html`.

[Ope12] OpenStack. Zuul - A Project Gating System, 2012. URL: `http://docs.openstack.org/infra/zuul/gating.html`.

[Ope15] OpenJDK. Jcov, jan 2015. URL: `https://wiki.openjdk.java.net/display/CodeTools/jcov`.

[Osh13] Roy Osherove. Build Pattern: Gated Commit, jan 2013. URL: `http://osherove.com/blog/2013/1/20/build-pattern-gated-commit.html`.

[Ple15] Christian Plewnia. A Framework for Regression Test Prioritization and Selection. Master's thesis, RWTH Aachen University, dec 2015.

[Pre10] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.

[RH96] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996. `doi:10.1109/32.536955`.

[RST+04] X Ren, F Shah, F Tip, BG Ryder, and Ophelia Chesley. Chianti: A prototype change impact analysis tool for Java. In *the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2004. URL: `http://sciris.shu.edu/masplas2004/MASPLASPapers/Paper7.pdf`.

[SRRT06] Maximilian Stoerzer, Barbara G Ryder, Xiaoxia Ren, and Frank Tip. Finding Failure-inducing Changes in Java Programs Using Change Classification. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 57–68, 2006. URL: `http://doi.acm.org/10.1145/1181775.1181783`, `doi:10.1145/1181775.1181783`.

[SSR10] Vibha Singhal Sinha, Saurabh Sinha, and Swathi Rao. BUGINNINGS: identifying the origins of a bug. In *Proc. ISEC*, pages 3–12, 2010. URL: http://portal.acm.org/citation.cfm?id=1730874.1730879, doi:http://doi.acm.org/10.1145/1730874.1730879.

[ŚZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes?, 2005. doi:10.1145/1082983.1083147.

[Tep15] Kirill Teplinskiy. Bisect plugin for TeamCity, 2015. URL: https://github.com/tkirill/tc-bisect.

[Wlo08] Jan Wloka. Chianti Project Page, 2008. URL: http://aria.cs.vt.edu/projects/chianti/.

[WS08] Chadd Williams and Jaime Spacco. SZZ revisited: verifying when changes induce fixes. In *2008 workshop on Defects in large software systems (DEFECTS '08)*, pages 32–36, 2008. URL: http://portal.acm.org/citation.cfm?id=1390826$\delimiter"026E30F$nhttp://www.cs.umd.edu/{~}pugh/ISSTA08/defects2008/papers/p32-williams.pdf, doi:10.1145/1390817.1390826.

[WZKC11] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and S C Cheung. ReLink: Recovering links between bugs and changes. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–25, 2011. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-80053202000{&}partnerID=40{&}md5=0bc9b3583d933d489020762df57b9c31, doi:10.1145/2025113.2025120.