Semantic Conflicts in Version Control Systems

Nuno Guilherme Nunes Castanho

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Dissertação orientada por:
Prof. Doutora Maria Antónia Bacelar da Costa Lopes
e co-orientada pelo Prof. Doutor José Carlos Medeiros de Campos
Acknowledgments

Throughout the development of this work I have received lots of support and assistance.

I would like to thank my supervisors, Professor Antónia and Professor José, for all the guidance, suggestions and trust they placed in me. Every feedback I received allowed me to push this work to the next level.

I would like to thank my parents, Lucília and Paulo, and my brother, Daniel, for the support I received without ever asking for it and for always being there for me.

I would like to thank my partner and girlfriend, Beatriz, for always pushing me to achieve more, for keeping me afloat when things seemed to be sinking, for hearing me ramble and complain to no end, and for everything she did for me throughout this work and the last years.

In addition, I would like to thank my friends and colleagues for all the moments and laughs we shared that allowed me to take a step back and clear my head, only to return stronger.

Finally, I would like to single out my oldest friends. Thank you for finishing this journey with me.
Para a minha tontinha...
Resumo

A realização de trabalho cooperativo em projetos de desenvolvimento de software está normalmente associada ao uso dum sistema de controlo de versões que permite trabalho paralelo sobre os mesmos ficheiros. Estas edições paralelas, no entanto, podem originar conflitos durante a integração das mudanças.

Certos tipos de conflitos são detetados e apresentados aos programadores automaticamente pelo sistema de controlo de versões. No entanto, existem conflitos que não são detetados automaticamente e introduzem defeitos no sistema.

Os conflitos de integração podem ser classificados como textuais, onde há mudanças distintas à mesma linha de código; sintáticos, que levam a sintaxe inválida no resultado da integração; estruturais, que são mudanças na estrutura que podem ser combinadas, de maneira significativa, de diversas formas; e semânticos, que levam a que o resultado da integração se compõe de maneira errônea.

As abordagens atuais para detetar conflitos semânticos assentam principalmente em análise estática de programas. Estas soluções são, porém, muito limitadas e, em particular, não permitem detetar conflitos que resultam de mudanças em classes diferentes.

Este trabalho foca-se na detecção de conflitos semânticos baseada na geração automática de testes unitários que revelem comportamentos indesejados. Esta geração automática de testes é guiada pela informação sobre as mudanças concorrentes.

Existe uma grande variedade de definições para conflitos semânticos na literatura. A definição adotada neste trabalho para este tipo de conflitos foi dividida nos conceitos de comportamento emergente e comportamento perdido. Cada um destes conceitos está definido à base de testes.

Especificamente, comportamento emergente representa o comportamento que emergiu no resultado da integração sem que nenhum programador o tenha escrito explicitamente. Comportamento perdido representa as mudanças introduzidas numa variante que se comportam de maneira diferente no resultado da integração.

Dizemos que existe comportamento emergente se, num cenário em que M é o programa que integra as mudanças nas variantes V₁ e V₂ duma base B, um teste t para M falha em V₁ e V₂ (quando aplicável). Dizemos que existe comportamento perdido quando um teste t para uma variante Vᵢ falha em B e em M (quando aplicável).
A solução concebida neste trabalho para detetar conflitos semânticos foi implementada na ferramenta UNSETTLE. UNSETTLE trabalha com quatro versões dum sistema (a base, duas variantes e o resultado da integração) e procura gerar testes que revelem comportamento emergente. A ferramenta calcula as diferenças entre a base e cada uma das variantes e compara as mudanças a um conjunto de padrões que capturam situações potencialmente causadoras de conflitos semânticos. Se as mudanças correspondem a um dos padrões, UNSETTLE tenta gerar um teste que revele o conflito guiada pela informação no padrão.

A representação mais comum de mudanças em ficheiros trabalha ao nível do texto, usando operações de inserção e remoção de linhas. Esta representação, no entanto, não é adequada para raciocinar sobre conflitos semânticos, nomeadamente para encontrar padrões de mudança relevantes. Por esta razão, foi desenvolvido, um modelo que permite definir padrões de mudança. Cada padrão é um tuplo \(< X, \phi, B, \Delta_1, \Delta_2 > \) onde: 

- \( X \) é o conjunto de variáveis para identificar os elementos, como classes e métodos;
- \( \phi \) é o conjunto de restrições das variáveis;
- \( B \) é a condição que a base tem de cumprir; e
- \( \Delta_i \) é um conjunto de ações num ramo. Cada padrão de mudança possui também um objetivo de teste que define qual a classe alvo para a geração, quais os métodos a cobrir num teste e quais desses métodos não devem aparecer diretamente no teste. Por exemplo, se tivermos dois métodos \( M \) e \( M_1 \), onde \( M \) depende de \( M_1 \) e ambos são modificados paralelamente, vamos querer que o nosso teste cubra \( M_1 \) passando por \( M \) para averiguarmos se as mudanças no primeiro afetam as mudanças no segundo.

Foram definidos 29 padrões de mudanças que capturam situações potencialmente causadoras de conflitos semânticos, nomeadamente situações relacionadas com sobrecarga de métodos, redefinição de métodos herdados e mudanças paralelas aos mesmo elemento. Alguns destes padrões representam variações da mesma mudança abstrata.

O UNSETTLE é constituído por dois componentes. Um componente, chamado de Pattern Matcher, é responsável por calcular as mudanças entre a base e cada variante e verificar se estas correspondem a um dos padrões de mudança. O outro, chamado Test Generator, é responsável por gerar um teste que revele o comportamento emergente.

O componente Pattern Matcher é também constituído por dois componentes. Um componente, chamado Change Instance Handler, é responsável por calcular uma instância da mudança para um padrão de mudança. Uma instância de mudança contém o estado da base e as mudanças entre a base e cada variante. O outro componente, chamado Matching Handler, verifica se a instância de mudança corresponde ao padrão para o qual foi construída.

O Change Instance Handler identifica o estado da base e calcula as diferenças entre a base e as variantes. Para isto, utiliza a biblioteca Spoon [1] para representar as árvores de sintaxe abstratas e o algoritmo GumTree [2] para obter a diferença entre duas árvores.

Depois de ser identificada a instância de mudança para um padrão, o Matching Han-
dler vai identificar os possíveis valores para cada variável do padrão. Para cada combinação de possíveis valores na instância, vai verificar se as condições se verificam. Se estas forarem verificadas, o componente obtém o objetivo de teste definido no padrão para iniciar a geração de um teste.

Para a geração automática de testes, foi estendida uma versão da ferramenta EvoSuite [3] que, através dum algoritmo genético, gera testes que satisfaçam um critério de cobertura.

Dada a natureza do nosso problema, queríamos gerar um teste para o resultado da integração que falhasse nas variantes. Começámos por adaptar a ferramenta para trabalhar com três versões dum uma classe. Visto que poderíamos ter de cobrir métodos sem que estes aparecessem no teste, estendemos a ferramenta para marcar estes métodos como sintéticos. Desta forma, sabemos que não vão ser adicionadas chamadas a estes métodos no teste, mas o caminho de execução pode passar pelos mesmos.

Para guiar o algoritmo evolucionário, foi criada uma função que otimiza testes que falham nas variantes e passam no resultado da integração. Esta função foi dividida em dois componentes. O primeiro aborda a cobertura dos métodos descritos no objetivo de teste. Visto que cobrir os métodos pode não ser suficiente para revelar o conflito, o segundo componente da função maximiza a diferença dos objetos no teste entre o resultado de integração e cada variante. A razão para esta abordagem assenta no facto de se conseguirmos atingir estados diferentes para o mesmo objeto, as probabilidades de revelar o conflito aumentam.

A avaliação da ferramenta focou-se em três questões diferentes: capacidade de detetar os diferentes tipos de conflitos (RQ1); capacidade de detetar conflitos em instâncias de integração recolhidas da literatura (RQ2); e escalabilidade da ferramenta para detetar um padrão (RQ3).

Para responder a RQ1, construímos 10 instâncias de integração a partir de projetos de unidades curriculares passadas onde injetámos um conflito. UNSETTLE conseguiu detetar o padrão de mudança em todas, mas só conseguiu gerar um teste que revelasse o conflito em oito com os valores padrão de configuração porque, nos outros dois casos, a única diferença entre as versões era um resultado booleano.

No contexto de RQ2, foram recolhidas da literatura 19 instâncias de conflitos semânticos. O UNSETTLE foi capaz de detetar os padrões em todos os casos, mas apenas conseguiu gerar um teste que revelasse o conflito em seis destas instâncias. Isto deveu-se ao facto de, durante a geração, a ferramenta não conseguir gerar objetos relevantes que permitissem alcançar o conflito ou revelar diferenças entre as versões.

Para responder a RQ3, usámos uma das instâncias recolhidas da literatura e fizemos variar o número de ficheiros alterados e o número de padrões a analisar. No que toca aos padrões, testámos primeiro com apenas o padrão que descreve mudanças paralelas a um método. Depois, juntamente com esse, considerámos os que descrevem mudanças
paralelas a um construtor e a um atributo. Finalmente, considerámos todos os padrões.

Como esperado, o tempo que UNSETTLE demorou a tentar encontrar uma instância dum padrão aumentou com o número de ficheiros. Foi detetado também um grande aumento de tempo quando os padrões de mudança a considerar usavam relações de dependência.

Apesar das evidências recolhidas indicarem que a abordagem é útil porque pode efetivamente detetar conflitos, a abordagem seguida neste trabalho enfrenta duas principais limitações. A primeira está relacionada com o facto do tempo dado à ferramenta para encontrar uma instância dum padrão pode não ser suficiente. A segunda remete para a geração de objetos relevantes onde geradores de testes podem ter dificuldade em gerar um teste que alcance os conflitos semânticos se estes estiverem por detrás de condições muito específicas.

Visto que UNSETTLE é configurável, é possível aumentar o tempo disponibilizado dependendo da situação. Por exemplo, disponibilizar um intervalo maior e deixar correr durante a noite. A segunda limitação está fortemente ligada ao estado da arte dos geradores de teste, mas pode-se tentar usar abordagens como reflexão para alcançar os conflitos semânticos.

**Palavras-chave:** conflitos semânticos, teste de software, geração de testes, conflitos de integração, deteção de conflitos
Abstract

Merging parallel changes is a common occurrence for developers working in collaborative software projects. Nowadays, however, developers still rely on tools that perform textual merge to detect possible merge conflicts. While practical and efficient, this merging approach fails to detect semantic conflicts, that is, concurrent changes that cause the merged result to misbehave. It is well known that software testing is a powerful method to check if a software system matches its requirements and build the developer’s confidence that it is defect-free. As such, tests can be used to detect wrong behaviour in the merged result. Despite their capabilities, however, tests might not cover the changed parts and, hence, do not reveal the bugs introduced during the merge. As such, the effectiveness of testing for conflict detection is directly dependant on the quality of the test suite being used. So far, the detection of semantic conflicts has been mainly approached through techniques that rely on the static analysis of programs and target specific types of concurrent changes. This work explores a different approach and proposes a solution to detect semantic conflicts that relies on automatic test generation and targets a variety of causes of such conflicts. The main idea behind the solution is to detect semantic conflicts as soon as possible, overcoming the dependency on the quality of the test suite by using automatically generated tests. The information about the concurrent changes that happened in the merge is used to guide the test generation. This led to the development of UNSETTLE, a tool capable of comparing the changes in different versions of a code base to a set of change patterns capturing known and common causes of semantic conflicts and automatically search for tests that reveal a semantic conflict. We performed a preliminary evaluation that showed us that UNSETTLE has the potential to be useful in detecting semantic conflicts. UNSETTLE was capable of generating a test that detected the emergent behaviour in 6 out of 19 cases of real semantic conflicts.

Keywords: semantic conflicts, software testing, test generation, merge conflicts, conflict detection
## Contents

| List of Figures | xiii |
| List of Tables  | xv  |

### 1 Introduction 1
1.1 Motivation ......................................................... 1
1.2 Problem ............................................................ 2
1.3 Approach ........................................................... 4
1.4 Goals and Main Contributions ........................................ 4
1.5 Structure of the document ........................................... 5

### 2 Related Work 7
2.1 Background .......................................................... 7
  2.1.1 Merge ........................................................... 7
  2.1.2 Merge Conflicts ............................................... 8
2.2 Related work ......................................................... 12
  2.2.1 Preemptive Conflict Detection .................................. 13
  2.2.2 Conflict Detection with Tests Generation ......................... 14
  2.2.3 Conflict Detection with Program Analysis ......................... 14
  2.2.4 Detection of program differences across versions .................. 15
  2.2.5 Change Detection, Change Models and Change Patterns ............ 16
  2.2.6 Code Analysis .................................................. 20
2.3 Summary ............................................................ 20

### 3 Solution Overview 23
3.1 Semantic Conflicts .................................................. 23
3.2 Solution .............................................................. 26
3.3 Summary .............................................................. 31

### 4 Change Patterns 33
4.1 Change Patterns Design ............................................... 33
4.2 Implementation ....................................................... 38
# List of Figures

1.1 Example of a Semantic Conflict. ........................................... 3  
2.1 Software Evolution. ................................................................. 8  
2.2 Octopus merge scenario. ........................................................... 10  
2.3 Diff of the **Counter** class and its variant as shown in Git. .......... 16  
2.4 AST representation of the **Counter** class (blue nodes) and a variant (red nodes). White nodes are common nodes to both. ......................... 17  
3.1 Merge scenario. ........................................................................ 23  
3.2 Representation of Example 1 of emergent and lost behaviour where the test applies to every version. ................................................... 24  
3.3 Representation of Example 2 of emergent and lost behaviour where the test does not apply to one version. ............................................ 25  
3.4 Representation of Example 3 of lost behaviour where the test does not apply to the base or merge versions. ................................. 26  
3.5 UNSETTLE, a tool for detecting semantic merge conflicts. .......... 27  
3.6 System Architecture. ................................................................. 30  
4.1 Another example of a Semantic Conflict. ..................................... 34  
4.2 Simplified structure of ConflictPattern ......................................... 39  
5.1 Pattern Matching Component decomposition. ................................. 42  
5.2 Representation of `sb.append("Hello World!")` obtained with GumTree. 43  
5.3 Objective of the ChangeInstanceHandler. .................................... 46  
5.4 Simplified structure of ChangeInstance. ...................................... 47  
5.5 Entity Processors. ................................................................. 47  
5.6 Delta Processors Hierarchy. ..................................................... 50  
7.1 Filtering of state-of-the-art merge instances. ............................. 67  
7.2 Complete filtering process. ..................................................... 68
List of Tables

4.1 The model’s entities and relationships. ..................................... 35
4.2 The model’s actions. ............................................................. 36

7.1 Groups and description for the constructed change patterns. ........... 65
7.2 Fabricated merge instances and the semantic conflict present. ........... 66
7.3 Results of UNSETTLE for the fabricated merge instances. .............. 66
7.4 Results of UNSETTLE for the fabricated merge instances without conflicts. 67
7.5 Semantic conflicts present in the collected merge instances. Note that the
merge instance of antlr4 contained the semantic conflict in two different classes. ..... 69
7.6 Results of UNSETTLE for the collected merge instances. Yes* – UNSET-
TLE had trouble generating a test that revealed the conflict but could succeed when ran
with a long time budget. No* – the conflict was not detected in the original code. .... 70
7.7 Results of testing the matching aspect of UNSETTLE by varying the
number of patterns and input files for the netty project. .......................... 72
Chapter 1

Introduction

This chapter provides the motivation for the development of this work, describes the context in which it was developed and presents its main objectives.

1.1 Motivation

Collaborative development of software systems is often linked with the use of a Version Control System (VCS) to help manage the evolution of software. Nowadays, these systems allow developers to create and work in different branches and with it parallel editing of the same software artefacts. However, the creation of branches can lead to conflicts between the concurrent changes. These are known as merge conflicts because they only become real during the merge process, i.e., when one developer wants to bring the changes made in one branch into the other one.

Some merge conflicts are automatically detected during the merge process and presented to developers who are asked to solve them. For instance, this is the case when two developers change the same line of code in two different ways. Whenever there are conflicts that are not detected and remain unnoticed, the merge process introduces software bugs\(^1\) in the code base.

In this way, in the context of collaborative software development, developers end up spending a lot of their time and effort resolving merge conflicts and fixing software bugs introduced by manual or automatic merge [4]. As reported by Lo et al. [5], the results of a survey conducted at Microsoft indicate that novel solutions for conflict detection and resolution are perceived as extremely useful in practice by developers.

The fact that there are software bugs that are introduced by the merge operation and end up in production code also justifies the relevance of conflict detection and resolution. For example, one of the suspected sources for the duplicate line in the Apple “goto fail”

---

\(^1\)Throughout this document we may use the terms bug to refer to a fault—a flaw in a component of the software system that caused the system to behave incorrectly (e.g., condition in an if statement)—and failure, a component of the software system that is not behaving as expected, to refer to both cases of errors and failures.
vulnerability is that of a merge that went wrong\textsuperscript{2}.

Not only do bugs cost millions for the companies responsible but they can also damage the environment and lead to the loss of life \cite{6}. With the increasing rate at which software and computer systems take part in our daily lives, from self-driving cars to electrical stations, it is more crucial than ever that bugs are detected and fixed before the system is deployed.

Being able to detect bugs that are introduced in the merge and signal them as merge conflicts is also quite important because it is well known that the sooner an error is detected the cheaper it is to resolve it. This is particularly true when talking about architecture and design errors but applies to software bugs as well.

1.2 Problem

Merge conflicts can appear in many forms \cite{7}. This work, however, is focused on the detection of the most difficult to address type of merge conflicts, *semantic conflicts* \cite{7}.

Textual conflicts are concurrent changes to the same line of code in different ways, e.g., one developer changes the step in a while loop to $i = i + 2$ while another changes it to $i += 2$. This type of merge conflicts is the one detected by modern VCSs like Git\textsuperscript{3} and Subversion\textsuperscript{4}.

Syntactic conflicts originate from merging textually compatible changes that result in code that is syntactically invalid. For example, one developer changes the signature of the method `calculateFactorial(int n)` to `factorial(int n)` while another adds calls to `calculateFactorial(int n)`. This type of conflict can be detected by using a compiler to check the validity of the merged result.

Structural conflicts come from refactoring and restructuring changes that do not change the semantics of the program but that can be combined, in a meaningful way, in different ways. As an example of a structural conflict, suppose we have a hierarchy where `Vehicle` is an abstract class with subclasses `Car` and `Truck`. One developer creates a new subclass `Bus` while another creates an intermediate class `HeavyVehicle` that becomes the superclass of `Truck`. In merging, the choice arises of whether `Bus` is made a subclass of `Vehicle` or `HeavyVehicle`. Most existing merge approaches do not consider this type of conflict \cite{7}.

Semantic conflicts are concurrent changes that cause the merged result to misbehave. For instance, as represented in Figure 1.1, one developer updating a method `distance()` that returned the Euclidean distance to return the Manhattan distance to the origin while another creates a new method `move()` that calls `distance()`, assuming it is the Euclidean distance. Introduced bugs, in a semantic conflict, can go undetected for a long time.

\textsuperscript{2}https://dwheeler.com/essays/apple-goto-fail.html (accessed June 2021)
\textsuperscript{3}https://git-scm.com (accessed June 2021)
\textsuperscript{4}https://subversion.apache.org (accessed June 2021)
time, making them harder to fix, or go undetected altogether and enter production, reaching the users and clients. Existing approaches to detect semantic conflicts have severe limitations and cannot be used in practice.

Existing approaches to conflict detection explore a variety of techniques, such as static analysis of program behaviour [8, 9, 10] and workspace awareness [11, 12, 13], and rely on different techniques for detecting and calculating program differences [14, 15, 16].

While approaches that aim to prevent conflicts are considered to disrupt productivity [11, 12, 13], the few that address semantic conflicts take too much time and inherit the limitations of the Satisfiability Modulo Theories (SMT) solvers they use [8, 9, 10], e.g., exponential running time. Moreover, most of them address only conflicts that arise from concurrent changes in the body of a procedure/method, for instance, two concurrent
optimisations that do not work together. Some other approaches only work with simple imperative programming languages [8, 14]. A detailed analysis of the state-of-the-art on software merging and merge conflicts can be found in Chapter 2.

1.3 Approach

Regardless of the type of software project, testing is a crucial part of any project’s lifecycle. Tests help developers detect bugs in the code as well as build confidence that the system and each of its components meets its design and behaves as intended.

Since tests are a powerful tool to detect software bugs, they can be used to detect bugs introduced during the merge process. However, the number of bugs detected depends on the quality of the test suite. Additionally, developers might only test their changes after a considerable amount of time has passed so as to not disrupt their current work, causing any fixes to be more expensive as they have to recall the changes they made.

The main idea that will be explored in this work is to overcome the dependency on the quality of the test suite being used by automatically generating tests to detect semantic conflicts. The information about the concurrent changes that happened in the merge will be used to guide the test generation.

More concretely, this work aims to develop a solution capable of detecting semantic conflicts in Java code during the merge process of two branches employing automated test generation techniques. Java is one of the most popular programming languages used around the world according to the TIOBE’s index and therefore a natural choice for our work. Moreover, as an Object Oriented (OO) language, Java code is prone to complex semantic conflicts related to features such as polymorphism, inheritance, and accessibility modifiers. These conflicts, if not detected, give rise to harder to detect software bugs. Java also happens to be the most popular choice in existing studies on merge conflicts and a popular programming language for Android mobile applications, opening another area for possible future development.

Thus, this work aims to develop a solution that improves the state of the art in the detection of semantic conflicts in Java code. With this solution, developers would be informed of the existence of software bugs introduced in the code base by the merge process. In this way, the detection of semantic conflicts would not be limited to the inherent quality of the test suite being used, namely whether it covers the code changes.

1.4 Goals and Main Contributions

Exploring the fact that tests are crucial in the discovery of semantic conflicts, this work plans to automatically generate tests that will reveal the existent semantic conflicts with a high degree of precision and probability.
The goal of the work is to identify some main sources for semantic conflicts in Java programs, develop techniques to represent those sources abstractly, and generate tests to detect those conflicts.

The main contributions of this work are:

- A new formulation of semantic conflicts based on tests.
- A model to represent patterns of change, appropriate for reasoning about semantic conflicts.
- An external DSL to define change patterns as per the developed model.
- A set of 29 change patterns that codify situations that are common causes of semantic conflicts.
- A tool to detect semantic conflicts by automatically generating tests guided by the information of the changes.
- An evaluation of the tool’s efficiency and scalability, and limitations of its approach.

### 1.5 Structure of the document

This document is organised as follows:

Chapter 2 presents the background, terminology and related work that showcase the different tools and techniques that have been proposed in the literature to detect semantic conflicts.

Chapter 3 presents the definition of semantic conflict adopted in this work and an overview of the solution.

Chapter 4 presents the representation of change patterns.

Chapter 5 presents the design and the implementation of our tool’s pattern matching component.

Chapter 6 presents the extensions to a test generator tool to fit our needs.

Chapter 7 presents the evaluation and results of our work.

Chapter 8 presents the summary of our work and provides some ideas to investigate in the future.
Chapter 2

Related Work

This chapter presents the notions and definitions in the literature regarding software merging and merge conflicts and the related work focused on the detection of semantic conflicts.

2.1 Background

2.1.1 Merge

Merging is a core operation in VCSs that integrates parallel variants of a file into a new version. In every merge operation, we have at least two variants of a software artefact that become the parents of the merged result or merged version. These parents share a common original version called the base version. In the end, these parents are variants of the base version. The variants being merged do not need to be direct descendants of the base version, meaning there could be more commits between them.

For example, in Figure 2.1 we have eight commits. Commit V7 is the merged result of merging V4 and V5 so V4 and V5 are V7’s parents. V2 is the base for V4 and V5 thus they are its variants. Identically, V8 is the merged result of merging V6 and V7 whose base is V2. V6 and V7 are V8’s parents and variants of V2. Most merge situations only use two variants as parents for the merged result. However, the latest version of Git supports octopus merge\(^1\) for merging more than two versions.

There are various merge approaches across the literature [7, 17, 18, 19] and the merge result and the detected conflicts depend on the one being used.

Two-way vs. Three-way merge. The first distinction in a merge is whether it is a two-way or three-way merge. While two-way merging only uses the information of the two variants to be merged, three-way merging also relies on the information in the base version [7]. In two-way merging, we do not have enough information to determine if the differences in the variants of the base are caused by simultaneous changes or not.

\(^1\)https://git-scm.com/docs/git-merge/2.32.0 (accessed June 2021)
It is important to note that three-way merging does not make the base version a parent of the merged result. During the merging process, the algorithm simply uses the information in the base version to determine if the differences between the variants are from parallel changes to the same element or not.

**State-based vs. Change-based merge.** State-based merging relies on the information (state) of the artefacts and not on how they were obtained. A two-way merge is also state-based as it does not rely on the information of how the variants were obtained [7]. Opposed to state-based, we have change-based merging that uses the information about the changes made to the artefacts. A three-way merge is also change-based. Operation-based merging is a flavour of change-based that models the deltas between versions as operations or transformations [7].

**Unstructured vs. Structured merge.** Merge techniques can also be distinguished on how they represent the artefacts [7, 17, 18, 19]. Tools that perform unstructured merge represent the software artefacts as plain text. This is the most common type of merge in VCSs because of its efficiency, scalability and accuracy [7]. Tools that perform structured merge use the information about the program’s structure and semantics to perform the merge. These techniques tend to reduce the number of reported conflicts, but they are quite slow. According to Apel et al. [18] in a recent paper, currently “no practical structured-merge tools for mainstream programming languages are available”.

There are also tools that use a combination of both worlds called semi-structured merge [17, 18, 19]. FSTMerge [17, 19] requires we provide the grammar of the programming language so the merge algorithm can take this information into account. However, the bodies of methods are represented as plain text and, hence, conflicts inside a method are not resolved automatically. JDime [18] performs unstructured merge, switching to structured merge when a conflict is detected.

### 2.1.2 Merge Conflicts

The need for merging comes from parallel changes to the same software artefacts. These changes can then lead to conflicts known as merge conflicts. Different notions of these
merge conflicts are present in the literature [7, 10, 11, 13], sharing some resemblances with each other.

To Brun et al. [13], any conflict that is textually safe to merge is a higher-order conflict while, as mentioned before, Mens [7] classifies syntactic conflicts as concurrent changes that are textually safe to merge yet result in invalid syntax in the merged result. Guimarães and Silva [11] present a similar definition for what they call language conflicts, that is, changes that result in compilation errors.

Guimarães and Silva [11] define structural conflicts as four possible conflict subtypes: pseudo direct conflict, attribute change & change conflict, node change & deletion conflict and inconsistent node type conflict. Keep in mind that the authors define these notions while modelling the system as a labelled node tree.

A pseudo direct conflict happens when concurrent modifications change the different attributes of the same node or the same attribute to the same value. The attribute change & change conflict occurs when an attribute of a node is changed to different values. A node change & deletion conflict comes from simultaneous modification and deletion to the same node. Finally, the inconsistent node type conflict happens when parallel modifications try to put nodes of different types at the same position.

Looking at each of the structural conflicts, we can see they relate to the definition of structural conflict proposed by Mens [7], corresponding to situations in which the merge algorithm does not know how to structure the result.

Another type of merge conflicts are test conflicts [11]. These conflicts are tests that fail in the merged result. This definition is a gateway to define semantic conflicts through tests like Ji et al. [20] and Silva et al. [21] do.

Kasi and Sarma [12] divide conflicts into three main categories: direct conflicts that happen because of concurrent changes to the same artefact; build failures that happen when different artefacts changed concurrently cause syntactic errors; and test failures that happen when concurrent changes cause different behaviour and result in test failures.

**Semantic Conflicts.** Like the other types of merge conflicts, different notions of semantic conflicts are adopted by different authors [7, 10, 12, 13, 20, 22].

Mens [7] defines semantic conflicts as changes that result in unwanted behaviour in the merged version, distinguishing them based on whether they can be detected during compile time. Semantic conflicts detected during compilation are called static semantic conflicts.

According to Marthin Fowler [22], semantic conflicts are concurrent changes that are safe to be merged at a textual and syntactic level but cause changes in the behaviour. Guimarães and Silva [11] use the same notion but call these conflicts behaviour conflicts. Kasi and Sarma [12] use the notion of indirect conflict as conflicts that occur because of changes in one file that affect the changes in another.
Brun et al. [13] define semantic conflicts as part of higher-order conflicts such as compilation errors or test failures. Pastore et al. [10] also use the term higher-order conflicts, stating that they occur when parallel changes to the code result in a faulty system that is textually safe to merge.

**Semantic Conflict Freedom.** Sousa et al. [9] do not define semantic conflict but present instead the concept of *semantic conflict freedom*. The key idea is that any change in the behaviour of the variants with respect to the base is significant and must be preserved in the merged result while the unchanged behaviour common to all three programs must also be preserved in the merged result. More concretely, given four versions of a program—Base, P₁, P₂ and M—where P₁ and P₂ are variants of Base and M is a possible merge of the two, semantic conflict freedom requires that if the execution of P₁ (or P₂) yields a different value for any variable v compared to Base, then M should reflect that difference and match P₁ (or P₂) on the value for v. In other words, semantic conflict freedom requires that:

- if P₁ and Base do not match on the value of v then M should match P₁
- if P₂ and Base do not match on the value of v then M should match P₂

In all the other cases, Base, P₁, P₂ and M must match on the value of v.

This definition of semantic conflict freedom directly relates to the notion of *interference* introduced by Horwitz et al. [8] which defines that two variants interfere with respect to a base if, after a computation from the same initial state, the value of a variable v is different in the base and the two variants.

**Unexpected and Lost Behaviour.** Inspired by semantic conflict freedom and relying on test cases with regression assertions, Ji et al. [20] define the notions of *unexpected behaviour* and *lost behaviour* in the context of octopus merging as follows:

- Let t be a test case for the merged program M. M is said to have some unexpected behaviour if, for every variant Pᵢ, the same assertion ψ of t is violated.
- Let t be a test case for any variant Pᵢ that fails in Base and M over the same assertion ψ. This means that the new behaviour introduced in the variant Pᵢ is missing after merging and, hence, configures a situation of lost behaviour.
As an example, consider the `Point` class shown in Listing 2.1 and the evolution depicted in Figure 2.2. Let us say that Anna, Bob, and Mike check out a copy of that class. Anna changes the implementation of the `distance()` method to calculate the Manhattan distance instead of the Euclidean distance to the origin and commits her changes, creating the commit $P_1$. Meanwhile, Bob changes the `move()` method to add to each coordinate the current distance to the origin and commits his changes, creating the commit $P_2$. Mike creates the `toString()` method for the string representation of a point and commits, creating the commit $P_3$. All changes are textually and syntactically compatible and can be easily merged as seen in Listing 2.2. However, this scenario presents unexpected and lost behaviour.

A test for the `move()` method over a point different from the origin with a regression assertion based on the behaviour of $M$, fails in $P_1$, $P_2$ and $P_3$. Failing in $P_1$ and $P_3$ is expected since the `move()` method has been updated (by Bob in $P_2$). However, it should not fail in $P_2$ since this commit was the one to change the method. It fails in $P_2$ because that version of the `move()` method was under the assumption that the `distance()` method still used the Euclidean distance, which is no longer true in $M$. The test for $M$ failing in $P_1$, $P_2$ and $P_3$ represents the unexpected behaviour that neither Anna, Mike nor Bob programmed but happened because of their concurrent changes.

```
public class Point {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public void move() {
        this.x += 1;
        this.y += 1;
    }

    public double distance() {
        return Math.sqrt(Math.pow(getX(), 2) + Math.pow(getY(), 2));
    }
}
```

Listing 2.1: Point class (Base).
A test for the `move()` method with a regression assertion based on the behaviour of $P_2$ reveals lost behaviour. It fails in `Base` as expected since Bob changed the method and it fails in the `M` because the test is expecting the result to be obtained with the Euclidean distance, which does not happen in `M`. This shows that the intended behaviour Bob introduced is missing in the merged result.

**Interference.** Very similar to the previous definition of lost behaviour, Silva et al. [21] propose the notion of interference which states that for four versions of a program—`Base`, $P_1$, $P_2$ and `M`—where $P_1$ and $P_2$ are variants of `Base` and `M` is the result of merging the two, the changes in $P_1$ interfere with the changes introduced in $P_2$ if a test $t$ for $P_2$ fails in the `Base` and `M`.

## 2.2 Related work

Several tools have been developed with the goal of detecting semantic conflicts, exploring different approaches. Some generate tests to detect different behaviour [20] while others
use specification mining [10] or static analysis [8, 9] for the same purpose. Additionally, tools such as WeCode [11] and Crystal [13], designed to improve awareness between developers, also perform conflict detection.

Only a couple of existing tools can detect conflicts beyond procedure/method level [11, 20], namely those that are caused by OO features such as inheritance and polymorphism.

### 2.2.1 Preemptive Conflict Detection

Proactive detection of conflicts has been investigated in different works exploring different ideas such as continuous merging, speculative analysis, and workspace awareness.

**WeCode.** This tool [11] is an awareness tool implemented as an Eclipse plugin. It performs continuous merging and presents detected conflicts to the developers. With continuous merging, any change, committed or uncommitted, is merged, compiled and tested in a special system called *merged system*. Both systems (the one under construction and the merged system) are modelled as labelled trees where each node has a type and attributes specified by the domain (e.g., Java). Dependencies between nodes such as the “extends” relationship are edges between involved nodes, forming a graph.

This tool has some limitations, however. It is currently limited to the VCS Subversion and to Eclipse Equinox\(^2\). Also, while capable of detecting semantic conflicts through testing, the test suite used is the one maintained by the developers of the project under merge which might not cover execution paths that reveal the semantic conflicts. As an awareness tool, WeCode might disrupt productivity by distracting developers.

**Crystal.** This tool [13] is also an awareness tool that tries to minimise the number of conflicts by performing the operations of the VCS in the background. However, unlike WeCode, it merges committed changes only. Using the relationship between the local repositories of two developers and the state of the repository itself, it suggests what actions can be made and their outcome. It can also suggest who can perform actions that change the relationship between repositories and who already has changes that resolve existing conflicts.

Much like WeCode, the test suite used is the one maintained by the developers of the project under merge and, as any awareness tool, might disrupt the developers’ work.

**Cassandra.** This tool [12] tries to avoid the conflicts by proactively suggesting task orders. It tracks the activities of the developer in the Eclipse Integrated Development Environment (IDE), such as “open” and “save”, the activities in the VCS and the activities in the issue tracker, tracking the task’s precedents and dependencies and the developer’s preference for the ordering of their tasks.

\(^2\)https://www.eclipse.org/equinox/ (accessed June 2021)
This tool is subjected to the downsides of awareness tools like WeCode and Crystal. Also, to use this tool’s full capabilities of task suggestion, teams must use the Mylyn\(^3\) Eclipse plugin.

### 2.2.2 Conflict Detection with Tests Generation

Testing On Merges (TOM) [20] is a tool that detects semantic conflicts in two-way and three-way merges and in octopus merges. The tool identifies the added and changed entities (as well as the ones affected by the changes) and generates regression tests using EvoSuite [3].

The tool generates tests for the merged version and runs them in its parents to search for unexpected behaviour. To search for lost behaviour, it generates tests for the variants and runs them in the merged and base versions. When the same assertion is violated in all the executions (and the generated test is stable), the test is added to the pool of tests that detect a semantic conflict and is shown to the developers.

In order to evaluate their tool, the authors constructed merge conflicts by mutating the source code and used no other conflicts (e.g., from open source repositories). Conflicts generated this way are by no means equivalent or representative of real-world scenarios which questions the tool’s capabilities with real-world merge situations.

Silva et al. [21] set out to answer if automatically generated tests could be used to detect their notion of interference (see Section 2.1.2) introduced during the merge process. They collected a total of 38 merge instances from the state-of-the-art and used EvoSuite [3], EvoSuiteR [23], and Randoop [24] to automatically generate tests.

Out of the 14 merge instances identified with an interference, they were able to generate tests that revealed the interference in two of those instances.

Silva et al. [21] also state that automatic generation of tests to detect semantic conflicts faces three main problems: generating relevant and significant objects that reach the problem code; generating relevant assertions that reveal the conflict; and generating assertions not only for the return values but also objects that were used by a method.

### 2.2.3 Conflict Detection with Program Analysis

Several static analysis tools have also been developed to help developers detect semantic conflicts.

**Integrate.** Horwitz et al. [8] present a tool that given three programs, where two are variants of the third, merges them if they do not interfere. Interference (see Section 2.1.2) is verified through static analysis of the program. If the variants do not interfere, there

\(^3\)https://www.eclipse.org/mylyn/ (accessed June 2021)
are no semantic conflicts and the merged result reflects the changes of the base that were defined in each variant. The unchanged behaviour of all three is also preserved.

The algorithm works with three versions of a program to detect interference but assumes that expressions only have scalar variables and that programs are only made up of assignment, conditional and while loop statements.

**SafeMerge.** Sousa et al. [9] developed a tool that verifies semantic conflict freedom (see Section 2.1.2). The tool generates a shared program with holes to represent where the versions differ. Each version is modelled as edits that describe how to fill each hole in the shared program to obtain the matching statement in said variant. Semantic conflict freedom is expressed with assertions over the return variables and modified global variables. The tool generates a relational post-condition that if true implies the assertions and uses an SMT solver to verify whether this happens. The tool uses dependency analysis on the shared statements and relational reasoning (seeing meaningful patterns) when it encounters a hole to generate the post-condition.

Since the tool relies on a SMT solver, it inherits its limitations. Moreover, SafeMerge focuses its analysis on one procedure at a time and assumes that callees invoked by it are semantic-conflict free. Also, as recognised by its authors, the analysis is only sound under the assumption the variants do not introduce non-terminating behaviour [9].

**Behavioral Driven Conflict Identification (BDCI).** This tool [10] detects semantic conflicts by generating the behavioural models of the different versions. The tool models the behaviour in pre and post-conditions, generating them through specification mining using Daikon [25]. To identify conflicts, it verifies if there is different behaviour between the variants and the base version. If for a given pre (or post) condition two variants are different among themselves and from the base, BDCI reports a semantic conflict.

The tool is limited to C programs and was evaluated with merges where changes affected at least 10 different files and affected 500 functions maximum (this second restriction was to avoid major redesigns of the system). The empirical study revealed it had a runtime of 31 minutes on average, making it suitable to be applied overnight and not after every merge.

### 2.2.4 Detection of program differences across versions

The detection of differences across programs has also been addressed in several works. Felsing et al. [14] developed a tool for regression verification that proves the behavioural equivalence of programs. This tool is limited by the fact that it only works with imperative programs with integer values.

Jackson, Ladd, et al. [15] present a tool for C programs that reports the semantic differences between two program versions, showcasing the changed dependencies in the
DiffGen [16] is a tool that generates unit regression tests to detect different behaviour between two versions of a Java class. After detecting the altered methods, DiffGen changes the visibility of the fields and creates a class (driver class) with one method for every changed method.

These methods receive an object of the older version and the required arguments for the method and create an object of the new version, deep copying the fields from the old version object into it. The rest of the method is the call of the altered method and the comparison of its result and the state of the two objects. With the commercial test generator JUnit Factory, DiffGen generates tests that will provide the arguments for the methods in the generated driver class. The tool cannot handle rename/refactoring of methods and signatures, and changes (delete, add, or modify) to the fields.

### 2.2.5 Change Detection, Change Models and Change Patterns

The calculation and representation of the changes operated on programs is also an important topic in the context of merge and conflict detection. Also relevant is the ability to represent change patterns.

The most commonly used model to represent changes operates at a textual level using simple operations of add a line and delete a line. This representation is obviously very lacking in terms of semantic changes since the artefacts are considered as simple text. An example of this can be seen in Figure 2.3. The code for the `Counter` class and its variant are in Listing 2.3 and Listing 2.4, respectively.

---

**Figure 2.3**: Diff of the `Counter` class and its variant as shown in Git.

```java
// Listing 2.3: Counter class.
public class Counter {
    int n;

    public void inc() {
        n++;
    }
}

// Listing 2.4: Counter's class variant.
public class Counter {
    private int n;

    public int inc() {
        return ++n;
    }
}
```
GumTree. As an interesting way to detect changes beyond the textual level, Falleri et al. [2] propose the GumTree algorithm to differentiate two Abstract Syntax Trees (ASTs), considering add, delete, update and move actions.

The AST considered in the approach consists of a labelled and ordered tree where the label of each node is the corresponding name of the production rule in the grammar. Each node has a value that can be empty. The differencing of two trees is based on edit actions where a sequence of such actions transforms the first AST in the second one. GumTree does not compute the shortest edit script but instead computes the one that reflects the developer’s intent [2].

In more detail, the edit actions used by GumTree are expressed as follows:

- `updateValue(t, v_m)` replaces the value of the node `t` with the new value `v_m`.
- `add(t, t_p, i, l, v)` adds the new node `t` with the label `l` and value `v` to the children of `t_p` as the `i`-th child. If `t_p` is null then `t` becomes the new root with the previous root as its child.
- `delete(t)` removes the leaf node `t`.
- `move(t, t_p, i)` moves the node `t` to the children of `t_p` as its `i`-th child.

In `update`, `delete` and `move` edit actions, the nodes `t` and `t_p` reference the original AST. In the `add` edit action, the nodes `t` and `t_p` reference the AST of the variant.

As an example, let us look at the simple `Counter` class shown in Listing 2.3 and its variant in Listing 2.4. We could represent these changes in edit actions as shown in the following sequence. The nodes are shown in Figure 2.4 where blue nodes exist only in the original class, the red ones exist only in the variant and white nodes are common to both.

```
add(t_1, t_2, 0, Modifier, private)
update(n_1, int)
add(t_3, t_4, 0, ReturnStatement, ε)
```
There have been works that tried to reduce the number of actions in a GumTree edit-script by considering *copy-paste* edit actions [26] and works that tried to improve the readability of said edit-scripts [27].

**Change Distiller.** Fluri et al. [28] also present the tool *ChangeDistiller* for source code differencing, using similar representation of change through edit operations. The authors also consider an *alignment* operation that moves a node among its parent’s children. The tool is available as an Eclipse plugin and relies on the VCS Concurrent Versions System (CVS)

> https://www.nongnu.org/cvs/ (accessed June 2021)

While more useful than a purely textual diff in terms of semantic changes, these models are too fine-grained, making them unsuitable for developers to read.

In another work, Fluri and Gall [29] name various change patterns, dividing them into *body-part changes* and *declaration-part changes*, classifying a total of 41 change types such as *Statement Insert*, *Statement Update*, *Increasing Accessibility Change* and *Return Type Insert*. However, these types are defined with the same edit actions (insert, delete, update and move) from before. For example, the *Statement Update* is $UPD(s, \text{val})$ where $s$ is the statement node.

**Coming.** Martinez and Monperrus [30] developed *Coming* with the purpose of mining Git commits through *change patterns*. While intended for repository mining, the tool can work at the filesystem level, i.e., with two source code files representing the before and after the changes. Each pattern contains a list of changes (e.g., insert) and the affected elements.

Listing 2.5: Example of a change pattern for Coming [30].

```xml
<pattern>
  <entity id="1" type="Method"/>
  <entity id="2" type="Parameter">
    <parent parentId="1" distance="1" />
  </entity>
  <action entityId="1" type="INS" />
  <action entityId="2" type="INS" />
</pattern>
```

[1]
Patterns, given as input, are represented as illustrated in Listing 2.5. The pattern provided as an example is the insertion of a method with one argument. In lines 2-5, we are saying there is a method node that is the direct parent of a parameter node. In lines 7-8, we are saying both of these nodes were inserted into the AST.

This way of representing changes is clearer to developers since it does not work at the AST-node level. However, since it was intended for repository mining, it does not support the representation of more sophisticated patterns of changes such as the insertion of a method that causes a situation of method overloading (the inserted method has the same name and number of arguments that an existing method and the types of all arguments are “compatible”).

Internally, Coming represents the change patterns as specified by Martinez et al. [31], i.e., as a tuple with three elements \( \langle L, R, U \rangle \) where \( L \) is a list of micro-patterns, \( R \) is the relationship map between the micro-patterns in \( L \) and \( U \) is the list of undesired micro-patterns. A micro-pattern is a tuple \((ct, et, pt)\) where \( ct \) is one of the 41 change types identified by Fluri and Gall [29], \( et \) is the entity type affected by the change and \( pt \) is the type of the parent where the change takes place. The entity type and parent type can be wildcards, represented by an “*”. For example, the insertion of an assignment statement inside an \( \texttt{if} \) block is represented by (“Statement insert”, “Assignment”, “If”).

As an example of a relationship between micro-patterns, consider a pattern with \( mp_1 \) and \( mp_2 \) as micro-patterns in \( L \), where \( mp_1 \) is an insertion of an \( \texttt{if} \) statement and \( mp_2 \) is an insertion of a \( \texttt{return} \). If we want to say that the \( \texttt{return} \) needs to be contained in the \( \texttt{if} \), we include in \( R \) the relationship \( mp_2.pt = mp_1.et \) between the two micro-patterns.

To decide whether a given pattern is present in a change, the tool first maps the AST changes to the micro-patterns of the pattern under consideration. In order to have a match, the change type, entity type and parent type must be equal. After the mapping, it verifies if the relationships established in the pattern are satisfied. Finally, it checks if no micro-pattern in the list of undesired micro-patterns is present.

**Refactoring Miner.** Tsantalis et al. [32] present the tool *RefactoringMiner*, capable of detecting refactoring operations between revisions. The tool is capable of detecting 40 refactoring types, such as *Rename Method* and *Move Class*, using a set of rules, expressed as logical conjunctions, to identify each type.

In order to identify in a revision \( r \) which refactoring type was made, the tool first extracts four sets of information. The first is \( TD_r \), which is the set of the type declarations (e.g., classes, interfaces, etc.). Each element in this set is a tuple \((p, n, F_r, M_r)\) where \( p \) is the parent of the tuple, \( n \) is its name, \( F_r \) is the set of fields and \( M_r \) the set of methods.

Every element in \( F_r \) is a tuple \((c, t, n)\) where \( c \) is the name of the type declaration that holds the field, \( t \) is the type of the field and \( n \) is its name. The elements in \( M_r \) are tuples \((c, t, n, P, b)\) where \( c \) is the name of the type declaration that holds the method, \( t \) is the
return type, \( n \) is its name, \( P \) is the parameter list and \( b \) is the body.

The fourth set is \( D_r \). This is the set of directories containing the modified files. Each directory is identified by its path \( p \).

The algorithm then matches code elements: a type declaration, field and method match another if the elements of the tuple match while a directory matches another if their paths match. Unmatched elements in the original version are put in sets \( TD \), \( F \), \( M \) and \( D \) respectively, representing possibly deleted elements, and unmatched elements in the new version are put in sets \( TD^+ \), \( F^+ \), \( M^+ \) and \( D^+ \) respectively, representing possibly added elements.

Finally, the algorithm tries to match the elements in these eight sets of unmatched elements using the refactoring rules to identify refactoring operations.

### 2.2.6 Code Analysis

**GumTree.** GumTree offers an Application Programming Interface (API) [33] to access the AST of a Java source code file and compute the differences between two ASTs. The nodes of the AST hold the information has primitive types where an integer identifies the type of node (e.g., MethodDeclaration, FieldDeclaration, etc.). Important information is kept as Strings as further explained in Chapter 5.

**Spoon.** Spoon [1] is a library for Java source code analysis and transformation. Its main goal is to allow developers to create domain-specific analyses and transformations on source code.

The Spoon metamodel is divided into three parts. The first is the structural part which holds and models the declarations such as classes, methods, interfaces, and other code elements. The second is the code part which models the executable code found inside methods. The third is the reference part which models the references to program elements such as a reference to a type.

All elements in the Spoon metamodel inherit from one called \( CTElement^5 \) that has a parent relationship (to itself), representing that one element is contained in another. Spoon’s API allows for querying specific elements such as getting all the assignments inside a method body and supports AST visitors.

### 2.3 Summary

In this chapter, in addition to providing some background relevant for this work, we survey a large bulk of related work that has been developed in the last two decades.

---

5\( CT \) means “compile-time” [1].
As we can see, there are various merge approaches available. The most common in today’s VCSs are based around three-way and state-based merging and perform unstructured merge, i.e., treat the files as simple text. This is the most common type of merge because of its efficiency, scalability, and accuracy.

Merge conflicts can be divided into four main categories: textual, syntactic, structural, and semantic. However, different notions for the same concept are present throughout the literature. This is particularly true when talking about semantic conflicts where we have notions such as behaviour conflicts, indirect conflicts, and higher-order conflicts. Thus, it becomes clear that we need a concrete definition for semantic conflicts, adapted to the context of this work. For instance, define semantic conflicts formulated in terms of tests.

Various techniques have also been developed to detect semantic conflicts. These techniques use approaches like static analysis and specification mining to detect the conflicts. However, despite the various approaches and tools presented in this chapter, none of them tackled the detection of semantic conflicts through the use of change patterns.

Current models used to represent change are lacking in terms of semantics and readability. As such, we need a way to represent changes that is easy to read and focuses on the semantics of the code.

In the next chapter, we propose a new formulation of semantic conflicts based on tests and the overview of our solution to detect these conflicts.
Chapter 3

Solution Overview

This chapter starts by introducing the notion of semantic conflict that was adopted in this work. Then, it provides an overview of our solution to automatically detect these conflicts.

3.1 Semantic Conflicts

As discussed in Section 2.1.2, the definitions of semantic conflicts vary across authors. Despite the various definitions among different authors, semantic conflicts are always related to the appearance of misbehaviour. Since our goal is to generate tests that reveal a conflict, it is convenient to pick a notion of semantic conflict formulated in terms of tests. Ji et al. [20] and Silva et al. [21]’s definitions are, hence, good candidates. However, we found they miss certain cases and need to be refined. In particular, they do not consider the cases where a test for one version of the program does not apply to the others, i.e., the test does not compile with that version. This is fairly common, for example, when creating a new branch to develop a new feature as the other branch will not have the new methods.

Similarly to Ji et al. [20], we base our notion in the emergence and loss of behaviour. Emergent behaviour corresponds to the behaviour that appeared in the merged version without any of the developers writing it explicitly. On the other hand, lost behaviour corresponds to the newly added behaviour in a branch that behaves differently in the merge version. The main difference between our definition and Ji et al. [20]’s is that we consider the situations where a test $t$ for one version does not apply to another.

![Merge scenario](image)

Figure 3.1: Merge scenario.
Emergent Behaviour: Let \( t \) be a successful test for the merged program \( M \) with \( V_1 \) and \( V_2 \) as its parents. We then say that \( M \) has emergent behaviour if \( t \) applies to both parents and fails over the same assertion \( \psi \) in both, or \( t \) applies to a single parent and fails.

Lost Behaviour: Suppose that the parent versions \( V_1 \) and \( V_2 \) have the common ancestor \( B \). Let \( t \) be a successful test case for \( V_i \). If \( t \) applies to \( B \) and \( M \) and \( t \) fails in both over the same assertion \( \psi \), or \( t \) does not apply to \( B \) and fails in \( M \), or \( t \) does not apply to \( B \) and \( M \), then we say that the behaviour introduced by \( V_i \) was lost.

Note that our definition of lost behaviour is very similar to the notion of interference introduced by Silva et al. [21] (see Section 2.1.2). The main difference between our definition and Silva et al. [21]’s is that we consider the situations where a test \( t \) for the variant does not apply to the base or the merge result.

In what follows, we present examples that illustrate these two notions. Consider again the Point class shown in Listing 2.1.

Example 1: Suppose that Anna and Bob check out a copy of this class. Anna changes the implementation of the distance() method to calculate the Manhattan distance instead of the Euclidean distance to the origin and commits her changes, creating the commit \( V_1 \). Meanwhile, Bob changes the move() method to add to each coordinate the current distance to the origin and commits his changes, creating \( V_2 \). Both commits are textually and syntactically compatible. However, this scenario presents emergent and lost behaviour.

If we use a test for the move() method with a regression assertion based on the behaviour of \( M \) over a point different from the origin, the test will fail in \( V_1 \) and \( V_2 \). Failing in \( V_1 \) is expected since the move() method has since been updated. However, it should not fail in \( V_2 \) since Bob was the one to change the method move(). It fails because Bob’s version of the move() method is under the assumption that the distance() method still returns the Euclidean distance, which is no longer true in \( M \). This test reveals...
emergent behaviour in $M$, i.e., behaviour that neither Anna nor Bob programmed but happened because their changes interfere.

If we use a test for the $\text{move()}$ method with a regression assertion based on the behaviour of $V_2$ and execute it in $B$ and $M$, it will fail in both. It fails in $B$ as expected since Bob changed the method $\text{move()}$ and it fails in $M$ because the test is expecting the $\text{distance()}$ method to calculate the Euclidean distance. This shows that the behaviour introduced by Bob is missing in the merge.

Example 2: Let us now consider that, instead of changing the $\text{move()}$ method, Bob added a new method $\text{area()}$ that calculates the area of a circle centred at the origin that contains the point in question. Let us not forget that Bob thinks that the $\text{distance()}$ method returns the Euclidean distance and that Anna has changed it to the Manhattan distance.

If we use a test for the $\text{area()}$ method over a point different from the origin with a regression assertion based on the behaviour of $M$, it does not apply to $V_1$ and fails in $V_2$. It does not apply to $V_1$ because Bob has just created the method and fails in $V_2$ because $M$ calculates the Manhattan distance while $V_2$ calculates the Euclidean distance.

Similarly, if we use a test for the $\text{area()}$ method now with a regression assertion based on the behaviour of $V_2$, it does not apply to $B$ (since the method was just created) and fails in $M$ because the test expects the Euclidean distance.

This shows once more that the merged version has emergent behaviour from the concurrent changes and that the intended behaviour Bob introduced is missing.

Example 3: A situation where a test for $V_1$ or $V_2$ does not apply to $B$ and $M$ is trickier but possible nonetheless. Let us look at the $\text{Counter}$ class in Listing 3.1 and consider that Anna and Bob check out a copy of the class.
Figure 3.4: Representation of Example 3 of lost behaviour where the test does not apply to the base or merge versions.

Listing 3.1: Counter class.

```java
public class Counter{
    private int n;

    public void inc(){
        n++;
    }

    public int getValue(){
        return n;
    }
}
```

Anna creates a new method `timesInc()` that gives the number of times the method `inc()` was called by returning the field `n` and commits her changes, creating commit `V1`. Meanwhile, Bob adds a new constructor `public Counter(int n)` that sets the starting value for the counter, creating commit `V2`. While safely merged at a textual and syntactic level, `M` has a bug. If we call the method `timesInc()` on any Counter object that did not start at zero, we will receive an incorrect number because Anna’s implementation of the method was built on the fact that the field `n` always started at zero.

If we use a test for the `timesInc()` method with a regression assertion based on the behaviour of `V1`, it does not apply to `B` nor `M`. It does not apply to `B` because Anna has just created the method and it does not apply to `M` because a test for `V1` will create a Counter object with the default constructor, which has since been removed in `M` because of Bob’s changes. This reveals that the intended behaviour Anna introduced in `V1` was lost.

### 3.2 Solution

Our solution to detect semantic conflicts is to automatically generate tests that reveal emergent behaviour. The generation of each test is guided by the information about the concurrent changes that happened in the merge. We developed the tool UNSETTLE to target known and common causes of semantic conflicts.
Figure 3.5: UNSETTLE, a tool for detecting semantic merge conflicts.

As shown in Figure 3.5, UNSETTLE (aUtomatic uNit teSt gEneraTion for semanTic conflIct dEtectIoN) takes four versions as input, i.e., the base, the two variants, and the merge, and produces zero or more tests as output. Each test reveals a merge conflict.

UNSETTLE starts by computing the differences between the base version and each variant. Then, it compares these changes to a set of change patterns that capture potential causes for semantic conflicts. If the changes correspond to an instance of a pattern, UNSETTLE tries to generate a test that reveals a conflict. UNSETTLE relies on a configuration file that allows for some customisable options.

UNSETTLE addresses the detection of emergent behaviour only. As such, the output tests are regression tests for $M$ that will fail in the parents (if applicable). Extending UNSETTLE to cover lost behaviour would mean generating tests for $V_1$ and $V_2$ that fail in $B$ and $M$. This is something to be explored in the future.

In order to exemplify how UNSETTLE works, let us consider again the situation where we have a Point class (shown in Listing 2.1) that follows the evolution depicted in Figure 3.1. Anna and Bob check out a copy of the class and change it concurrently.

Anna once more changes the implementation of the distance() method to calculate the Manhattan distance to the origin and commits her changes, creating the commit $V_1$. Meanwhile, Bob changes the move() method to add to each coordinate the current distance to the origin and commits his changes, creating $V_2$.

Both of Anna and Bob’s changes are textually and syntactically safe. However, as we have seen before, this scenario presents a situation of emergent behaviour.

Let us say that we feed UNSETTLE with these four versions of the Point class ($B$, $V_1$, $V_2$ and $M$). UNSETTLE then starts by computing the differences between $B$ and $V_1$ and between $B$ and $V_2$. 
Anna’s changes will be replaced as a simple update to the `distance()` method and Bob’s as the insertion of a dependency to `move()`. As such, for the difference between $B$ and $V_1$ we will have something like “update `Point.distance()`” and for the difference between $B$ and $V_2$ we will have “insert dependency to `Point.distance()` in `Point.move()`”.

These changes are an instance of a change pattern that captures a common cause for a semantic conflict, i.e., one branch updates a method while another inserts a new dependency to that same method. As such, UNSETTLE will try to generate a test that reveals a semantic conflict.

For this particular case, UNSETTLE might generate a test like the one shown in Listing 3.2.

Listing 3.2: Example of a test that reveals the emergent behaviour.

```java
@Test
public void test0() {
    Point p0 = new Point(3, 4);
    p0.move();
    // p0 becomes (4, 5) in $V_1$
    // p0 becomes (8, 9) in $V_2$
    // p0 becomes (10, 11) in $M$
    assertEquals(10, p0.getX());
    assertEquals(11, p0.getY());
}
```

If we run this test in $V_1$, the first assertion will fail. This happens because the `move()` method in this version only increases each coordinate by one. As such, the `getX()` call will return 4 as opposed to the expected 10.

If we run this test in $V_2$, the first assertion will also fail. This happens because this version of the `move()` method adds the Euclidean distance to the origin to each coordinate. As such, we will be left with the point $(8, 9)$ from adding 5 to each coordinate. In the end, the `getX()` call will return 8 as opposed to the expected 10.

If we run this test in $M$, there is no failure. This happens because this version of the `move()` method adds the Manhattan distance to the origin to each coordinate. As such, we will be left with the point $(10, 11)$ from adding 7 to each coordinate, which matches the expected values.

Since the generated test passes in $M$ but fails in $V_1$ and $V_2$, it reveals that the changes performed by Anna and Bob resulted in the emergence of new and unintended behaviour.

In addition to this type of change pattern, UNSETTLE targets other common situations that often result in semantic conflicts such as parallel modifications to the same entity (e.g., method), method overriding and the removal of an overriding, and method overloading.

The change patterns used by UNSETTLE are read from a catalogue. In order to al-
low developers to easily define these patterns, we developed a Domain Specific Language (DSL). In this DSL, change patterns are defined in terms of a condition, two sets of actions, constraints, and a testing goal. The condition must hold in the base, while each set of actions expresses the delta between a revision and the base. This DSL is further explained in Section 4.1.

For instance, the change pattern considered before where one branch updates a method while another inserts a new dependency to that same method can be defined as shown in Listing 3.3

Listing 3.3: Change Pattern in our DSL.

```
Name: "Dependency Based (new dependency)"
Variables: $A $B $M $M1

Constraints:
$A can be equal to $B

Base Condition:
Class $A
Class $A has method $M
Class $B
Class $B has method $M1

Delta:
Update method $M of class $A

Delta:
Insert dependency to method $M in method $M1

Testing Goal:
Target Class: $B
Cover Methods: $B.$M1 $A.$M
Execute Methods: $A.$M
```

This change pattern starts by defining which variables exist (line 2). These variables will reference the code entities like classes or methods. By default, every variable has a different value. However, the constraint tells us that variables $A$ and $B$ can be equal. Note that in the pattern itself every variable is prefixed with a $.$.

In the base condition, we are saying there are two classes, $A$ and $B$ (lines 8 and 10, respectively). Because of the constraint defined in line 5, these variables can reference the same class. We are also saying that each of these classes has one method, i.e., class $A$ has method $M$ while class $B$ has method $M1$ (lines 9 and 11, respectively).

The action in the first delta (line 14) tells us that method $M$ was updated, while the action in the second delta (line 17) tells us that a dependency to it was added to method $M1$.

Finally, the testing goal tells us three things. The first is that the target class for the test generation is class $B$ (line 20). The second is that the generated test must cover both
methods $M_1$ and $M$ (line 21). The last one is that $M$ must be covered but not called in the test, i.e., a call to the method $M$ cannot appear directly in the test (line 22).

**System Architecture.** Our goal is to detect semantic merge conflicts by automatically generating a test that reveals the conflict. This generation is guided by the information about the changes between the base version and its variants. Hence, the system starts by identifying what changes were made between the base version and each of its variants and check if the change instance matches any of the change patterns known to result in a semantic conflict. If a change instance that matches one of the patterns is found, the system tries to generate a test that reveals that conflict.

Figure 3.6 shows the decomposition of the system in two components. The *Pattern Matcher*’s responsibility is to identify the state of the base, the changes between the base version and each of the variants, and check if the change instance matches one of the change patterns.

If the *Pattern Matcher* detects an instance of a change pattern, it passes the testing goals of that pattern to the *Test Generator*. The *Test Generator*’s responsibility is to create a test that reveals the emergent behaviour.

The *Pattern Matcher* uses the Spoon [1] API to represent the ASTs and the GumTree algorithm for Spoon [34] to compute the edit actions between two ASTs. The test generation tool used for the *Test Generator* component is an extension of EvoSuite [3] called EvoSuiteR [23]. The changes performed to extend EvoSuiteR to suit our needs are described in Chapter 6.
3.3 Summary

We based our notion of semantic conflict in *emergent* and *lost* behaviour. On one hand, emergent behaviour represents the behaviour that appeared (or emerged) in the merge version without any of the developers writing it. On the other hand, lost behaviour is the newly added behaviour that was introduced in a branch that is missing or behaves differently in the merge version.

UNSETTLE, our tool for semantic conflict detection, targets the appearance of emergent behaviour. As such, given four versions of a system (the base version, the two variants, and the merge), it will generate a test that passes in the merge version and fails in both variants if applicable.

UNSETTLE uses the information about the concurrent changes to guide the test generation by comparing them to a set of change patterns. These patterns represent concurrent changes that often result in semantic conflicts. UNSETTLE starts by trying to match the changes between the base and each of its variants to the change patterns. If there is a match, it uses the information in the pattern to know which class and methods to target during test generation.

In the next chapter, we present the design of our change model to represent changes and the DSL to define change patterns.
Chapter 4

Change Patterns

This chapter discusses the representation of change patterns that capture potential causes of semantic conflicts and presents the DSL we developed to support the definition of these patterns.

4.1 Change Patterns Design

One of the base concepts for this project is the use of change patterns to represent potentially conflicting changes in two revisions of a given Java source code file. As such, we need an appropriate model to represent the changes.

As discussed in Section 2.2.5, the models that have been used to represent change between two revisions of a Java file are either too generic (e.g., line diff), too fine-grained for developers to read (e.g., GumTree [2]) or lack expressiveness (e.g., Coming [30]).

The change model needs to be independent of the AST and needs to abstract from changes that are meaningless in the context of semantic conflicts. For example, we might be interested in knowing that a method \texttt{m()} was edited but not that the edit itself was the insertion of a \texttt{for} loop. However, we might be interested in knowing specific types of changes inside a method like the insertion of a method call or new field access.

It is important to note that to detect semantic conflicts it is not enough to look at the deltas between the two revisions of a file. Whether a pair of deltas causes a semantic conflict depends on their common ancestor.

As an example, consider the semantic conflict depicted in Figure 4.1. In this case, the first delta would be something like \textquote{insert method resize() in Square”} and \textquote{insert access to field size in resize()”} and the second delta would be \textquote{insert field size in Square”}. From these deltas, we have no way of knowing that the class \texttt{Square} is a subclass of \texttt{Shape} and that \texttt{Shape} has a field \texttt{size}. These two facts are essential to conclude that the changes in the two deltas potentially interfere.

For representing patterns of change that potentially interfere and result in a semantic conflict, we consider tuples of the form \(< X, \phi, B, \Delta_1, \Delta_2 >\) where:
Chapter 4. Change Patterns

Figure 4.1: Another example of a Semantic Conflict.

- $X$ is the set of variables used in the pattern. These variables identify the different entities, such as classes or methods.
- $\phi$ is a set of variable constraints.
- $B$ is a condition the common ancestor must match. This condition characterises the relevant entities that are present in the common ancestor.
- $\Delta_i$ is a set of edit actions. This set represents the edit actions performed in a branch.

Normally, two variables can be equal unless stated otherwise. However, we followed the opposite approach and, by default, every variable is different. This was because, during the conceptualisation of changes that resulted in semantic conflicts, the number of cases where different variables could have the same value was very small.

As mentioned before, base conditions only need to address some aspects of the code. In Table 4.1, we show the ones we considered important based on the entities involved in the mutation operators introduced by Offutt et al. [35].

An extends relationship between two classes, i.e., $\text{Class } A \text{ extends class } B$, means class $A$ is a subclass of class $B$. An implements relationship between a class $A$ and an interface $I$ means that class $A$ implements the interface $I$. The has relationship between a class $A$ and a method $M$ means class $A$ contains a method $M$. The same thing applies for the has relationship between a class and a constructor or field. The does not have relationship between a class $A$ and a method $M$ means the class $A$ does not contain a method $M$.

We only included the does not have relationship between a class and a method (and not also fields and constructors) to be able to differentiate from the methods that were inherited but not overwritten and the ones that were implemented by that class.

The depends relationship between methods, i.e., $\text{Method } M \text{ depends on method } N$, means method $M$ depends, directly or not, on method $N$. The compatible relationship rep-
Table 4.1: The model’s entities and relationships.

<table>
<thead>
<tr>
<th>Entities</th>
<th>Class</th>
<th>Interface</th>
<th>Method</th>
<th>Constructor</th>
<th>Field</th>
<th>Access Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>extends</td>
<td>implements</td>
<td>has does not have</td>
<td>has</td>
<td>has</td>
<td>-</td>
</tr>
<tr>
<td>Interface</td>
<td>-</td>
<td>-</td>
<td>has</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Method</td>
<td>-</td>
<td>-</td>
<td>depends compatibile</td>
<td>-</td>
<td>reads writes</td>
<td>public, private, protected, package</td>
</tr>
<tr>
<td>Constructor</td>
<td>-</td>
<td>-</td>
<td>depends</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Field</td>
<td>has type</td>
<td>has type</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

represents a situation of method overloading. However, to avoid using the term “overloads”, we labelled it compatible. The reads or writes relationship between a method M and a field F means that the method M uses the value of F or writes the value of F, respectively.

The depends relationship between a constructor and a method, i.e., Constructor C depends on method M is identical to the depends relationship between methods. We chose not to include field accesses in constructors because they are almost always present in them.

The has type relationship between a field and a class, i.e., Field F has type A means that the type of the field F is class A. The same principle applies to the has type relationship between fields and interfaces.

The access modifier means that the entity has an access modifier (visibility). This is optional, i.e., we can define a method without specifying its visibility, and the possible values are public, private, protected, or package (no modifier).

We focused the entities and relationships on the ones we deemed relevant. However, adding more relationships to Table 4.1 would not be complicated on a technical level.

Using the entities and relationships described in Table 4.1, we then thought about which actions for each one were possible and relevant. These are shown in Table 4.2.

When it comes to classes, we only consider insert class actions. This is because we could not think of any situation where the removal of an entire class would lead to a semantic conflict. Most of the time, it would likely lead to a textual or syntactic conflict.

The insert method action is the insertion of a method in a class. Similarly, the insert field and insert constructor actions are the insertions of a field and a constructor in a class, respectively.

The insert dependency, insert visibility, and insert field read/write actions are, respectively, the insertion of the dependency, visibility modifier and field read/write in the entity of the table. For example, the insert dependency action in the Method line means we can insert dependencies in methods. Similarly, the insert visibility in the Field line means we can insert a visibility modifier in a field.

The delete actions follow a similar principle. For example, the delete method action
### Table 4.2: The model’s actions.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Insert Actions</th>
<th>Delete Actions</th>
<th>Update Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Insert class</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Interface</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Method</td>
<td>Insert method</td>
<td>Delete method</td>
<td>Update method</td>
</tr>
<tr>
<td></td>
<td>Insert dependency</td>
<td>Delete dependency</td>
<td>Update dependency</td>
</tr>
<tr>
<td></td>
<td>Insert visibility</td>
<td>Delete visibility</td>
<td>Update visibility</td>
</tr>
<tr>
<td></td>
<td>Insert field read/write</td>
<td>Delete field read/write</td>
<td></td>
</tr>
<tr>
<td>Constructor</td>
<td>Insert constructor</td>
<td>Delete constructor</td>
<td>Update constructor</td>
</tr>
<tr>
<td></td>
<td>Insert dependency</td>
<td>Delete dependency</td>
<td>Update dependency</td>
</tr>
<tr>
<td></td>
<td>Insert visibility</td>
<td>Delete visibility</td>
<td>Update visibility</td>
</tr>
<tr>
<td>Field</td>
<td>Insert field</td>
<td>Delete field</td>
<td>Update field</td>
</tr>
<tr>
<td></td>
<td>Insert visibility</td>
<td>Delete visibility</td>
<td>Update field type</td>
</tr>
</tbody>
</table>

is the removal of a method from a class, the *delete dependency* is the removal of a dependency from a method or constructor, the *delete visibility* is the removal of a visibility modifier from an entity, and the *delete field read/write* is the removal of one field access from a method.

We can apply the same principle for the *update* actions. The *update method* action is the edition of a method, the *update dependency* action is the change of one dependency to another, the *update visibility* action is the change of one visibility modifier to another, and the *update field type* action is the change of a field’s type to another.

**Listing 4.1: Parallel Changes change pattern**

```plaintext
1 Name: "Parallel Changes"
2 Variables: $A $M
3
4 Base Condition:
5 Class $A
6 Class $A has method $M
7
8 Delta:
9 Update method $M of class $A
10
11 Delta:
12 Update method $M of class $A
13
14 Testing Goal:
15 Target Class: $A
16 Cover Methods: $A.$M
17 Execute Methods:
```

Keeping in mind that the goal of using change patterns is to help guide the automatic test generation, we also include in our change patterns what we called a *testing goal*.

The testing goal requires three things. The first is the target class, i.e., the class we
will generate a test for. Then, come the methods the test needs to cover.

It is important to point out that just because a method is specified in the testing goal as needed to be covered, it does not mean a call to that method should appear in the test. In fact, there are certain situations where we do not want a method to be called directly from the test. For example, if we have two methods, M0 and M1, and M0 depends on M1 and both were edited in different branches, we will want our test to reach M1 by means of M0. A test that directly calls M1 is not able to reveal the interference of the changes in M0.

As such, the third thing a testing goal needs is to know which of the methods to cover only need to be executed, i.e., are not to be called directly from the test.

To make it easy for UNSETTLE users to include new patterns, we developed an external DSL that supports the directives described in this section and the definition of change patterns.

These patterns are defined by starting with a name. This is merely for the sake of readability. Then, as per our tuples, they specify a set of variables to identify the entities.

Listing 4.2: Dependency Based change pattern

```
Name: "Dependency Based (new method)"
Variables: $A $B $M $M1

Constraints:
$A can be equal to $B

Base Condition:
Class $A
Class $A has method $M
Class $B

Delta:
Update method $M of class $A

Delta:
Insert method $M1 in class $B
Method $M1 depends on method $M

Testing Goal:
Target Class: $B
Cover Methods: $B.$M1 $A.$M
Execute Methods: $A.$M
```

Optionally, after the variables, come the additional constraints to specify which variables can have the same value. After it, we have the base condition that characterises the common ancestor. After the base condition come the two deltas, each one relating to $\Delta_1$ and $\Delta_2$ respectively. These deltas are the set of edit actions performed in each branch. Finally, we have the testing goal as described above. Examples of our change patterns are shown in Listing 4.1 and Listing 4.2. These patterns target different situations that can happen during the merge process. For instance, parallel changes to the same entity, either
a constructor, field or method (as shown in Listing 4.1), the insertion of a dependency to a method $M$ while this method is updated (as shown in Listing 4.2), or method overriding and method overloading.

Listing 4.3: Dependency Based change pattern variation

```plaintext
Name: "Dependency Based (new class)"
Variables: $A$ $B$ $M$ $M1$

Base Condition:
Class $A$
Class $A$ has method $M$

Delta:
Update method $M$ of class $A$

Delta:
Insert class $B$
Class $B$ has method $M1$
Method $M1$ depends on method $M$

Testing Goal:
Target Class: $B$
Cover Methods: $B$.{$M1$} $A$.{$M$}
Execute Methods: $A$.{$M$}
```

It is important to point out that many of these patterns reference the same abstract change pattern. For example, when we insert a new dependency, we can do so by adding it to an entirely new class, to a new method, or to an already existing method. An example of such variation between change patterns is shown in Listing 3.3, Listing 4.2, and Listing 4.3.

The reason for this variation in different patterns comes from the fact our DSL does not have logical connectors, i.e., “or” and “and”. These were not considered because we tried to simplify the patterns given the constraints we had. This was a poor choice of design.

The fact we have multiple patterns for the same abstract change means we also have very similar patterns with the same testing goal, as seen with the patterns shown in Listing 3.3, Listing 4.2, and Listing 4.3. This is something that can be improved in the future.

Improving and redesigning this DSL is something to be explored in future work.

### 4.2 Implementation

In order to translate these patterns into Java objects, we implemented a parser for the DSL detailed in Section 4.1. This way, users specify the patterns according to the grammar of our DSL and do not have to be concerned with how to build the respective objects they represent.
To represent our change patterns and use them within UNSETTLE, we implemented one class for each of the entities represented in Table 4.1, each one with the same relationships that are shown in the table.

Each of these classes has a field that represents its variable and were all named following the same convention, i.e., the entity they represent, followed by the word “Pattern”. For example, we implemented classes such as MethodPattern and ClassPattern.

A similar approach was taken to handle the actions described above. For every action, we have one class with the words “Pattern” and “Action” at the end. For example, our update method action translates to the UpdateMethodPatternAction class.

Unlike the entity pattern classes, our action pattern classes do not have a field that represents a variable. Instead, they reference the entity pattern classes that are relevant for them. For example, our UpdateMethodPatternAction class contains a MethodPattern that represents the method to be updated and a ClassPattern that is the holder of that method.

The action pattern classes were divided according to the operation, i.e., insert, delete and update. Each one was made to extend InsertPatternAction, DeletePatternAction, or UpdatePatternAction depending on its operation. The exception to this were the actions pertaining to access/visibility changes that were all encapsulated in a single class (VisibilityPatternAction).

Each of these patterns (entities or actions) knows how to verify if it matches with a respective instance. For example, a MethodPattern knows if it matches a MethodInstance and an UpdateMethodPatternAction knows if it matches an UpdateMethodAction.

If we look at the patterns displayed in Section 4.1, we will see that our base conditions are made up of either class or interface patterns. These class patterns then hold the fields, constructor and method patterns that hold other patterns of their own. To this, we call the BasePattern.

Very similarly, our DeltaPatterns are composed of ActionPatterns. In the end, a Con-
ConflictPattern is nothing more than a BasePattern and two DeltaPatterns. A simplified structure of the class is shown in Figure 4.2.

4.3 Summary

In order to represent the changes that happened in a merge in a relevant way for semantic conflicts, we designed our change patterns as $< X, \phi, B, \Delta_1, \Delta_2 >$ tuples. These tuples describe one or more variables that characterise the common ancestor and the edit actions performed in each branch. Additionally, our change patterns also have a testing goal, i.e., what class and methods to target.

For code entities, our change patterns can specify classes, interfaces, methods, constructors, and fields. Around those entities, we can specify insert, delete, and update actions.

To make it easy for users to write change patterns, we developed a DSL. With it, they can design change patterns by specifying which code entities, such as classes and methods, exist in the base version and what changes to those entities might result in semantic conflicts.

In the next chapter, we present the design and implementation of the pattern matching component of UNSETTLE that tries to match the changes between the base and each of its variants to the change patterns that often result in semantic conflicts.
Chapter 5

Pattern Matcher

This chapter presents the design and the implementation of our pattern matching component.

5.1 Design

Given that the main goal of the pattern matching component is to identify the changes between the base version and each of its variants and to match them against a set of change patterns, this component is decomposed as shown in Figure 5.1.

The first step in our process is to identify a change instance for each change pattern in the catalogue of patterns. This responsibility falls upon what we call the Change Instance Handler. After we have computed a change instance for a change pattern, we need to try and match it with the change pattern it was meant for. This responsibility falls under what we call the Matching Handler.

A change instance represents what happened in the merge. On one hand, it describes the state of the common ancestor, i.e., what classes, methods, etc., existed. On the other hand, it also describes what actions, if any, were made to those entities.

It is important to point out that the patterns are used to calculate a change instance. This happens because we are interested in building a change instance that has what is important for the pattern in question.

For example, if we were building a change instance for the pattern shown in Listing 4.1, we would only need to be concerned with classes and methods because these are the only entities used in the pattern. There would be no need to check each method for its dependencies because the pattern is not worried about dependencies between methods. This search for dependencies, however, would be important if we were analysing the pattern shown in Listing 4.2 since the pattern is based on that relationship.

A similar process can be applied for the actions that appear in the change instance. The pattern shown in Listing 4.1 is not worried whether a new dependency is inserted in a method. Therefore, that particular change would be translated into the update method...
action instead of the *insert dependency to method*.

We have to keep in mind that, even if a pattern specifies the *does not have* relationship for a method, we still want to build our change instance with methods. The fact the pattern has a *does not have* relationship only means it is worried about methods. Thus, we will build our change instances with the entities and actions that are used in the change pattern.

### 5.1.1 Change Instance Handler Design

Before we can identify our change instance, we need to represent the ASTs of the classes inside our system.

As discussed in Section 2.2.6, GumTree [2] offers an API capable of representing the AST of a Java source code file and capable of computing the differences between two ASTs. This API, however, is not suitable for the purpose at hand.

On one hand, since the ASTs are not built according to the visitor design pattern, it is difficult to identify the different types of nodes. On the other hand, each node holds the information as primitive or String types, making it impossible to identify certain aspects like the type of the target in a method invocation. The example presented in Figure 5.2 illustrates the AST representation obtained for a simple invocation of the StringBuilder `append(String)` method like `sb.append("Hello World!")`.

The first problem could be solved by visiting the tree iteratively and checking the type of each node to decide how to proceed. The second one, however, cannot be fixed so easily.

These two problems can be overcome by using the Spoon [1] library. The Spoon
library represents ASTs in a way that easily allows developers to analyse its elements: nodes allow visitors (as per the visitor design pattern) and each node holds information that is dependant on its type. For example, method nodes know the signature of the method and invocation nodes know the type of the target.

With this API and the GumTree algorithm designed for Spoon trees [34], we are able to process an AST and the changes between two of them. These responsibilities are assigned to different components.

Before we can start the identification of the change instance, we have to load the source code files of the three versions of the system under test and create the Spoon objects that we will analyse. Then, for each change pattern, we will build a change instance. As mentioned before, this change instance will only have entities and actions that are relevant for the pattern under test.

We have one component responsible for identifying the base instance, i.e., the state of the common ancestor, called the Base Instance Handler, and another one, called the Delta Instance Handler, responsible for computing the difference at the AST-level between two files and then build the delta instance, i.e., the set of edit actions.

The Base Instance Handler transverses the Spoon objects and builds the base instance. The Delta Instance Handler obtains the GumTree edit actions between two versions of an AST and builds a delta instance.

In the end, our Change Instance Handler will have constructed a change instance for a particular pattern. This change instance will have a base instance and one delta instance for each branch.

This change instance, and its respective change pattern, are then passed to the Matching Handler that will calculate the possible combinations and check if there is a match. A general description of the algorithm is shown in Algorithm 1.

5.1.2 Matching Handler Design

Given a change instance for a given change pattern, the next step is to try to match it with the pattern it was meant for. This responsibility falls under the Matching Handler.

A change pattern as the one shown in Listing 4.1 can be seen as a canvas with holes. The variables need to be filled with actual values so that the different conditions hold.
Algorithm 1: Pattern Matcher algorithm.

```
patternMatching(baseFiles, v1Files, v2Files)
  for pattern in patternCatalog do
    changeInstance ← getChangeInstance(pattern, baseFiles, v1Files, v2Files)
    matches(changeInstance, pattern)
  end
getChangeInstance(pattern, baseFiles, v1Files, v2Files)
  baseInstance ← getBaseInstance(baseFiles, pattern)
  for i: 0 .. baseFiles do
    baseFile ← baseFiles[i]
    v1File ← v1Files[i]
    v2File ← v2Files[i]
    deltaV1 ← getDeltaInstance(baseFile, v1File, pattern)
    deltaV2 ← getDeltaInstance(baseFile, v2File, pattern)
    changeInstance ← <baseInstance, deltaV1, deltaV2>
  end
  return changeInstance
matches(changeInstance, pattern)
  varValues ← identifyVarsPossibleValues(changeInstance, pattern)
  generator ← combinationsGenerator(varValues)
  while hasNext(generator) do
    assignment ← next(generator)
    if holds(varCondition(pattern), assignment) and
      holds(baseCondition(pattern), assignment, changeInstance) and
      holds(deltaConditions(pattern), assignment, changeInstance) then
      testingGoalInstance ← instantiate(testingGoal(pattern))
      out! testingGoalInstance
    end
  end
```

The first step is to identify the possible values for each variable. Then, the conditions of the pattern are evaluated in the change instance with each combination of possible values for the variables. If all conditions hold, then we have found an instance of the change pattern.

The next step in our workflow is to generate a test that reveals the conflict. The assignment pairs of \((\text{variable}, \text{value})\) only have meaning during this matching phase and, as such, are useless from this point forward.

As mentioned before, our patterns have a \textit{testing goal}. If we look at our pattern in Listing 4.1, we will see this same testing goal defined with variables too. This means that during the assignment step of the values, these holes in the testing goal will also be filled. If we do find an instance of one of our change patterns, we will also find the class and methods we need to test in order to reveal the conflict.
5.2 Implementation

This section describes how each of the previous components was implemented.

5.2.1 Spoon Handler

As mentioned before, we have to “build” the different versions of the system under test. Since some of our change patterns are concerned with dependencies, we cannot just take into consideration the modified classes, but also the various classes that depend on them or are their dependencies.

To start the “construction” of the system under test, we first load all the classes into a launcher. This launcher is a class from the Spoon [1] API. As such, we will not go into much detail (more information about the API here\(^1\)). The launcher will allow us to “build” our system under test. By building, we mean create the Spoon objects representing each class, method, field, etc., in the source code and link them. These links can be extends relationships, invocations between methods, and other types of relationships that a code entity can have with another.

Since we are working with three versions of the system under test, we will have to load three different launchers.

After the loading step comes the building step. As described above, this step will create the various Spoon objects that represent the code entities in our system under test.

Once the system is built, we do not need to take every class into consideration. In fact, we are only concerned with the classes that depend on the modified classes or the ones that these classes (the modified ones) depend on.

As such, after we have “built” the system under test, we obtain these dependencies and dependants. Any class that was modified, is dependent on one of the altered classes or is a dependency of one of them is of interest to us. Any interface implemented by these classes or by the altered ones is of interest as well.

These classes and interfaces are the ones that will be passed to the ChangeInstanceHandler to build our domain objects and calculate the deltas.

It is important to point out that, because of the nature of the Spoon [1] launcher, there cannot be two classes with the same qualified/canonical name, even if these subclasses are in separate modules of the system. To bypass this, we need to be more specific with the paths used in the configuration file.

5.2.2 Change Instance Handler Implementation

As mentioned in Section 5.1.1, the purpose of the ChangeInstanceHandler is to identify the base and delta instances for a specific change pattern.

\(^1\)https://spoon.gforge.inria.fr/ (accessed June 2021)
At this point in our workflow, we already have the Spoon [1] objects that are of interest to us but we still need to represent them more appropriately to our problem. Basically, we want to process the Spoon objects and GumTree diffs into something we can work with as shown in Figure 5.3.

Much like we did for the pattern side of our system, we created one class to represent each entity described in our DSL shown in Section 4.1. This time, however, each object of these classes represents a real instance of that entity in the system.

Following a similar approach as the one described in Section 4.2, for each possible entity we created a corresponding class, adding the word “Instance” at the end. For example, we implemented the \texttt{ClassInstance} and \texttt{MethodInstance} classes.

A similar principle applies to the possible actions, except we do not add the word “Instance” at the end. For example, for the \textit{update method} action, we will have the \texttt{UpdateMethodAction} class.

Much like the pattern actions, the actions were divided according to their operation, i.e., \textit{insert}, \textit{delete}, and \textit{update}. Each action extended \texttt{InsertAction}, \texttt{DeleteAction}, or \texttt{UpdateAction} depending on its operation. The exception to this were the actions pertaining to access/visibility changes that were all encapsulated in a single class (\texttt{VisibilityAction}).

A simple structure of the \texttt{ChangeInstance} class can be seen in Figure 5.4.

\section*{Base Instance Handler Implementation}

The \textit{Base Instance Handler’s} responsibility, as mentioned in Section 5.1.1, is to identify the \textit{base instance}. By identity, we mean to go through the previously gathered Spoon [1] objects and process them into our equivalent types.

\section*{Processors} In order to build each object type (\texttt{ClassInstance}, \texttt{InterfaceInstance}, \texttt{MethodInstance}, etc.), we need to analyse the corresponding Spoon [1] object and, based on the
change pattern being tested, extract the relevant information.

With that in mind, we created what we call *processor classes*. Each of these classes’ responsibility is to convert one object from the Spoon API into our equivalent type.

We implemented one of these classes for each code entity we are interested in processing (ClassProcessor, InterfaceProcessor, MethodProcessor, FieldProcessor, and ConstructorProcessor).

Each of these classes implements the `Processor<K, V>` interface as shown in Figure 5.5. This interface requires a single method, labelled `process`, that receives an object of type `K` and returns one of type `V`. In our case, the `Ks` are Spoon objects and the `Vs` are our corresponding types.

For example, the `ClassProcessor` class implements `Processor<CtClass, ClassInstance>`. Its `process` method receives a `CtClass` object (the Spoon object representing a class) and returns a `ClassInstance`. On the other hand, the `MethodProcessor` class implements `Processor<CtMethod, MethodInstance>`. Its method receives a `CtMethod` and returns a `MethodInstance`.

A `ClassProcessor` uses a `MethodProcessor` to process methods, a `FieldProcessor` for fields and a `ConstructorProcessor` for constructors.
In order to avoid processing the same element more than once, we implement a basic caching mechanism. This caching process is not as straightforward as just reusing the same object because, as we mentioned before, we want to build these objects as per what is important for the pattern.

Listing 5.1: Parallel Changes Constructor change pattern.

Whenever a processor is called to process a Spoon object, it first checks if that object has already been processed. If it has not, it processes it and stores the resulting object. If that Spoon object has already been processed, the processor then checks if the instance object built has everything the change pattern needs. If something important for it is missing, it builds that fragment and adds it to the resulting instance.

For example, say we have a class $A$ with a method $M$. Now, we will process class $A$ as per the change pattern shown in Listing 5.1. For this pattern in particular, we are not interested in methods. Therefore, any $\text{ClassInstance}$ object built during the analysis of this pattern will not have any methods. Let us say that the processing of $A$ results in the $\text{ClassInstance} A_i$.

Now, after that, let us say that we are processing the class $A$ as per the change pattern shown in Listing 4.1. When processing $A$ as per this pattern, the $\text{ClassProcessor}$ will see that a $\text{ClassInstance}$ for $A$ has already been built, in this case, $A_i$. Fetching $A_i$, it will then check if it has any methods constructed since this change pattern is interested in methods. Since $A_i$ does not have any methods, the $\text{ClassProcessor}$ will hand $M$ to a $\text{MethodProcessor}$, receiving the $\text{MethodInstance} M_i$, and then add $M_i$ to $A_i$, updating the saved value and returning it.

This building process means that our objects might start small and grow as more change patterns are analysed.
**Delta Instance Handler**

As mentioned before, the purpose of the *Delta Instance Handler* is to compute the deltas between the different versions of the modified files and, after it, build the respective actions in our equivalent types.

In order to compute the differences beyond textual level, we used the GumTree algorithm [2]. Since we are working with the Spoon [1] API, we used the version of the algorithm that works with such objects [34].

GumTree computes the *edit actions* that turn one AST into another (see Section 2.2.5 for more information). These *edit actions* contain the performed operation (insert, delete, update or move) and the affected Spoon AST node.

For example, if the difference between two revisions of a class was the addition of a new method, we would have an object with the *insert* operation and a `CtMethod` object that represents the added method. Similarly, if the difference was a simple addition of a `for` loop, we would have an object with the *insert* operation once more and a `CtFor` object that represents the new loop.

**Delta Processors**  As mentioned in Section 2.2.6, the Spoon [1] API allows for the visitor design pattern. Because of it, we can easily process the different types of Spoon nodes.

Very similarly to the *processors* implemented for the *Base Instance Handler*, we created what we call a *delta processor* for each of the operations (insert, delete, update, and move). This resulted in the `InsertActionsProcessor`, `DeleteActionsProcessor`, `UpdateActionsProcessor` and `MoveActionsProcessor` classes. Each of these classes implements the Spoon interface `CtVisitor` to be able to visit the AST nodes and is responsible for creating our action objects (e.g., `UpdateMethodAction`).

These delta processors are also responsible for hiding unimportant changes to the code. For example, we are not interested in knowing that a `for` loop was inserted in a method and we want to treat this change as an update to said method.

Whenever the action and affected Spoon node are not of our interest, these delta processors hide that fact and replace it with something relevant for our purpose.

For instance, let us say that the difference between two versions of the same class was the insertion of a `for` loop. Since it was an *insert action*, the responsibility to process it falls upon the `InsertActionsProcessor`. This delta processor then knows that whenever it visits a `CtFor` (Spoon object representing a `for` loop) node it is supposed to create an `UpdateMethodAction` for the method that received the loop.

The above example shows us that these delta processors will not necessarily create an action that matches their name, i.e., the `InsertActionsProcessor` does not always create insert actions.

If we look at our DSL in Section 4.1, we will see that we do not consider *move*
operations important so it might seem odd that we have a `MoveActionsProcessor` class. As explained before, this class simply knows how to process operations that the GumTree algorithm considered as “moves” and translate them to relevant actions.

For example, the moving of a variable declaration within the body of a method should be considered an update to that method. As such, in that situation, the `MoveActionsProcessor` would create an `UpdateMethodAction` object.

While we consider insert, delete, and update access modifier/visibility actions, these types of actions are not processed by the delta processors described above. This is because the objects that represent the access modifiers are not visitable as per the visitor design pattern like, for example, `CtMethod` is.

In fact, the GumTree algorithm for Spoon [34] wraps these changes in a custom `CtWrapper`. To know the entity whose visibility was affected, we have to analyse the object of this wrapper class. Because of this, we implemented one delta processor for each type of visibility action we consider, i.e., `VisibilityInsertActionsProcessor`, `VisibilityDeleteActionsProcessor`, and `VisibilityUpdateActionsProcessor`. None of these classes implements the `CtVisitor` interface and, instead, have a visit method of their own that does the processing.

All of the delta processors used by the `Delta Instance Handler` extend an abstract class called `DeltaProcessor` as shown in Figure 5.6.

Since the `visit` methods imposed by the `CtVisitor` interface are `void`, the `DeltaProcessor` class has the field that will hold the resulting action. Not only that, but it also has utility methods that are helpful for all the delta processors. These methods use the processors described in Section 5.2.2 in order to convert, for example, a `CtMethod` into a `MethodInstance`.

After each GumTree operation has been processed, the resulting action is added to the `DeltaInstance` that was being created.

Since we transform certain operations into actions we are interested in, we can end up with duplicates. For example, say that in one branch two different changes were made to
a method and that these changes were the addition of a new variable and the addition of a
for loop. As explained before, each of these changes will result in an UpdateMethodAction.
There is no need to keep two UpdateMethodAction objects for the same method in
the same DeltaInstance since, as far as we are concerned, they represent the same change
in the code. As such, any duplicates are ignored when added to the DeltaInstance.

5.2.3 Matching Handler

As mentioned in Section 5.1.2, the Matching Handler’s responsibility is to check if the
ChangeInstance built by the Change Instance Handler is an instance of the change pattern
it was built for.

We can look at these change patterns as a canvas with holes. These holes represent our
variables. In order to check if a ChangeInstance constitutes a semantic conflict, we need
to fill these holes and then check if our now-filled canvas appears in the ChangeInstance.

In order to fill the holes, we first need to identify the possible values for each of
the variables. For this, we created what we call the variable identifiers. Each variable
identifier implements an interface called IVariableIdentifier that requires only one method
called identify. The identify method receives a ChangeInstance and a ConflictPattern
object and returns a map. This result is the mapping between the variables that appear in
the ConflictPattern and a list of possible values for each one from the ChangeInstance.
With this interface, we created one class responsible for each entity type of our domain.
For example, the ClassVariableIdentifier identifies the class variables and their possible
values while the MethodVariableIdentifier does the same but for methods.

Once we have identified the possible values for each variable, we test the possible
combinations of these values and see if an assignment of those values is present in the
ChangeInstance. It is important to point out that, unless two variables are identified as
being allowed to be equal, we do not test assignments where two different variables have
the same value. We also perform a “fitness” evaluation of the ChangeInstance before
we even start the identification of the variables and their possible values. We say that a
ChangeInstance is fitted for a ConflictPattern if it has the entities and the actions that the
pattern requires.

For example, there is no point in checking if a ChangeInstance that did not insert a
new method matches one of our method overriding patterns. The same principle applies to
checking if a ChangeInstance that only has classes with the default (still not overwritten)
constructor matches our parallel changes to a constructor pattern (see Listing 5.1).

Since the matching process can take a while, the time each change pattern is tested
for matching is one of the configurable options of UNSETTLE. It can take UNSETTLE a
while to verify a match because the depends relationship can have a lot of instantiations.

We could try to improve this process by using the changes to find possible values for
the variables. For example, if a method was updated, we would consider that method as a
possible value for the method variables. From that method, we would discover a possible
value for the class variables.

This approach is something we did not explore given our constraints but is something
to be explored in the future.

5.3 Summary

In order to represent the ASTs of the source files, we used the Spoon [1] library. This
library allows developers to easily analyse source code.

We can divide our pattern matching component into two others. The first, the Change-
InstanceHandler, is responsible for obtaining the change instance for a change pattern.
This component uses the GumTree algorithm for Spoon [34] to obtain the changes at the
AST-level. It is important that we use the change patterns when building the change in-
stance because we want to build an instance that is relevant to the pattern. If methods are
not important for the pattern, there is no need to build an instance with methods.

The second component, the MatchingHandler, is responsible for checking if a change
instance matches the change pattern it was built for. This component starts by identifying
the possible values for each variable in the pattern. Then, it assigns the possible combina-
tions and checks if that assignment exists in the change instance. If the MatchingHandler
finds a match, it uses the information in the change pattern to obtain the testing goal and
pass it to the test generator.

In the next chapter, we present the extensions made to the open-source test generation
tool EvoSuite [3] to fit our needs and generate a test that reveals emergent behaviour, i.e.,
passes in the merge but fails in its parents.
Chapter 6

Test Generation

For the automatic generation of tests, UNSETTLE uses an extension of EvoSuite [3] called EvoSuiteR [23] that searches for regression tests that reveal altered behaviour between two versions of a Java class.

This chapter provides an overview of EvoSuite and discusses how UNSETTLE uses testing goals to guide test generation.

6.1 EvoSuite and EvoSuiteR

**EvoSuite.** EvoSuite [3] is an automatic unit test generator for Java classes. The tool is based on an evolutionary algorithm that optimises test suites with the goal of satisfying one or more coverage criteria. Each chromosome in the evolutionary algorithm contains the test case being evolved. The tool uses various minimising fitness functions to evaluate the value of each test case. Each of these fitness functions is related to a test coverage criterion. For example, there is a fitness function that tries to optimise line coverage and another that tries to optimise branch coverage.

EvoSuite starts its evolutionary algorithm by generating a random population of test suites. These test suites are then evaluated with the fitness function being used and assigned a value that represents how good they are. The most fitting individuals (test suites) then suffer mutations or crossovers (combine two individuals). After that, EvoSuite evaluates the new individuals and repeats the process.

EvoSuite performs the evolutionary algorithm until it finds a test suite that satisfies the fitness function being used, i.e., the value of the fitness function for the suite is 0, or until the time budget runs out. After generating a test suite for a coverage criterion, EvoSuite generates regression assertions for each test in that suite.

**EvoSuiteR** EvoSuiteR [23] is an extension of EvoSuite that tries to generate a regression test that reveals that two versions of the same class have different behaviour. The tool assumes that the changes were not renaming of methods and that all other classes remain
unchanged.

Instead of using one of the standard coverage criteria and fitness functions, EvoSuiteR uses a custom, multi-purpose fitness function whose goal is to reach, infect and propagate (as per the Reachability, Infection, Propagation (RIP) model [36]) the different behaviour between the two versions. The RIP model states that to reveal a fault we need to satisfy three conditions [36]. First, we need to reach the locations in the code that contain the fault (reachability). Then, after executing the fault, the state of the program needs to be incorrect (infection). Finally, this incorrect state must cause the output of the program to be incorrect as well (propagation).

This fitness function is divided into three major aspects. The first one, called structural coverage, relates to the reachability of the code and tries to maximise the level of branch coverage of the two versions under test. The second aspect, called state difference, addresses the propagation of the different behaviour. This aspect tries to maximise the state distance between the objects of both versions. The computation of this state difference, which is based on the work of Ciupa et al. [37], calculates how different two objects are by analysing how distant the values of common fields are and how far apart the two objects are in the class hierarchy. The third aspect, called control-flow distance, addresses the chances of infection and tries to maximise the difference between the execution flows of the test in both versions.

Since EvoSuiteR works with two versions of a class, it needs to use two different classloaders. Thus, its chromosomes contain two test cases, one for each classloader.

Since the most recent version of EvoSuite does not include EvoSuiteR, we used an older version\(^1\). It is noteworthy that, since then, a lot of new features have been added to EvoSuite and the tool has gone through a major redesign. Adapting the most recent version of EvoSuite to work with our extensions is something to be addressed in future work.

### 6.2 Scope of the Extensions

Recall that we want to be able to generate a test that reveals the emergent behaviour in the merge version. So, we need to generate a test \( t \) that passes in the merge but fails in the parents that it is applicable.

EvoSuite works with fitness functions to evaluate how good each test is. As such, since we were not worried about the traditional coverage criteria, we had to create our own fitness function that optimised tests that revealed the semantic conflict.

Since our test generation is guided by a testing goal that tells us which class to target, which methods should be covered, and which of those methods should not be called in the test, we have to make sure the generated test complies with these constraints. We want to

\(^1\)https://github.com/EvoSuite/evosuite/tree/1.0.6 (accessed June 2021)
make sure that every method that needs to be covered is covered and that every method that should not appear in a test does not.

The RIP model tells us that reaching a method is not enough to reveal a fault. As such, we need some other measure, besides reaching the methods that we need to cover, to reveal the semantic conflict. We used the state difference measure already employed by EvoSuiteR to try to guide the tests to reveal the conflict.

The thought process behind this choice was that, if we could obtain different states in the three versions for the same object, the chances of revealing a conflict would increase.

Much like EvoSuiteR, we were not working with a single version of a class but three. So, one of the extensions we made was to create a new chromosome containing three test cases, one for each classloader.

Sometimes, the return values of the method we want to cover might be enough to reveal the semantic conflict. However, other times, the conflict might only happen at the state-level, i.e., it is not the returned value that is different. To reveal these semantic conflicts that are only detectable if we inspect the state of the objects in the test, we needed a way to obtain an \textit{hashCode}-like value that characterises each object so that we could generate an assertion for it. We called the method that calculates this value \texttt{allFieldsMethod}.

Once the test generation phase finishes, and before we generate the assertions for the test, we inject a call to the \texttt{allFieldsMethod} for each non-primitive, non-enumerate variable in the test. We insert these method calls at the end of the test to catch the difference that resulted from the operations performed during its execution.

### 6.2.1 Cover Methods

The first thing we tackled was how to provide EvoSuite with the class and methods that we needed to cover.

EvoSuite has an option to designate the target class for the test generation so we did not have to change anything regarding this aspect. It also has an option to specify a specific method that we would like the test to cover. This feature, however, did not serve our purposes. First, we might have multiple methods we want to cover in a test. Then, these methods might not be accessible by EvoSuite, i.e., they might be private. We could tell EvoSuite to use reflection to access private methods directly, but we do not know if the semantic conflict in a private method is only reachable when called from a public one.

As such, we implemented a new input option that we labelled as \texttt{cover\_methods} that tells EvoSuite which methods must be covered by the test.
6.2.2 Third Classloader

The main reason that we worked with EvoSuiteR was to generate tests that revealed different behaviour between versions of a class.

As mentioned before, the chromosome used by EvoSuiteR contains two test cases, one for each version’s classloader. As such, we implemented our own type of chromosome that is similar to the one from EvoSuiteR except it has three test cases. We called it *MultiTestChromosome*.

Whenever we run the test under evolution for the merge version, we also run it for the other two versions. Each result of these three executions is kept in our chromosome.

Since we were using completely different chromosomes from the standard ones, we also developed a chromosome factory responsible for creating *MultiTestChromosomes*.

6.2.3 Assertion Generation

In software testing there is the concept of *mutation testing*. Its goal is to evaluate a test suite by creating mutants of the code under test. These mutants are created by applying a single change that introduces a fault.

A mutant is killed if one test case in a test suite has different results from the original code. These mutants are used to evaluate how sensitive a test suite is. If a large number of mutants is not killed, the test suite is unlikely to reveal real faults.

EvoSuite, by default, keeps the assertions in the test that kill more test mutants. Other options include keeping the assertions that assess return values or simply keeping all the assertions it can generate. We developed a strategy that is a mix of the last two.

Once the test generation is complete, EvoSuite generates every possible assertion for each statement. From these, we keep the ones that assess return values but remove assertions that reference more than one variable. This is to avoid situations like the one shown in Listing 6.1 where we compare each new variable with previous ones.

Listing 6.1: Test example with multiple asserts.

```java
@Test
public void test0() throws Throwable {
    Point point0 = new Point(3, 4);
    int int0 = point0.getX();
    assertEquals(3, int0);

    int int1 = point0.getX();
    assertEquals(3, int1);
    assertEquals(int0, int1);

    int int2 = point0.getX();
    assertEquals(3, int2);
    assertEquals(int2, int0);
    assertEquals(int2, int1);
}
```
6.3 Fitness Function

For the purpose of generating a test that reveals emergent behaviour, we need to design a new fitness function that targets specific aspects.

Reachability. In order to reveal a semantic conflict in the code, we first have to be able to reach the portion of the code that contains the conflict. In our case, we had a set of methods that we wanted our test to cover.

State Difference. Since reaching the method that contains the semantic conflict is not enough to reveal it, we need to be able to generate tests whose objects reach different states when ran in different versions.

The first aspect we had to tackle was the reachability aspect. Let us start with the methods that we do not want to be called directly from the test. From here on, let us call these methods we do not want to appear in the test as secondary methods.

EvoSuite performs a certain degree of instrumentation before starting the evolutionary algorithm. One example is, for instance, to collect the lines of each class so that it can know if a test covers them or not when using the line coverage criterion.

When EvoSuite is generating tests and selecting which method call to insert next, it will ignore any method that is marked as being synthetic. With this information, we thought it best to simply mark all the secondary methods as synthetic during instrumentation. This does not prevent the methods from being called during the execution but prevents EvoSuite from directly calling such methods from the test itself.

As such, our fitness function only has to be concerned with knowing if a test covers the methods or not because, if a test covers one of the secondary methods, we know it is not called directly from the test.

During evolution, we can have access to the last result of each individual (test) and, among other things, can ask for the trace of method calls. One alternative to solving our reachability problem could be to simply query this trace of method calls and see how many of the methods we want to cover were already covered.

While simple, this solution is very lacking because we are simply working with binary results. Either one method is covered, in which case we added a zero (0), or a method is not covered, and we added a one (1). With this approach, we do not give EvoSuite any information about how close it truly is to cover the methods, we only tell it if it has covered them or not.

We used a different strategy which is able to inform EvoSuite how close it is to cover any given method. In order to do this, we consider the distance to reach any method $M$ as the distance to reach any of its lines. A similar approach was already used in the line coverage criterion.
Now, reaching a method does not necessarily mean we cover the conflict. We need a measurement, besides method reachability, to aid EvoSuite in generating a test that will fail in the merge’s parents.

Adapting the state difference aspect to our problem, we calculate the state difference of the objects in the test between the merge version and each of its parents. We then have to decide how to use these two values.

One option, and probably the most obvious one, would be to use the average between the two distances and maximise it. This, however, does not work because we want a test that will fail in both parents and not just one.

Imagine that for a test $t$, the state difference between $M$ (merge version) and $V_1$ (first parent) is 0 and the difference between $M$ and $V_2$ (second parent) is 1. This gives us an average of 0.5. If the state difference between $M$ and $V_1$ is 0, it means that there is no difference between the objects in the test, so it is not a very good test for our situation.

Now, imagine that we have a test $t_1$ where the state differences between $M$ and $V_1$ and $M$ and $V_2$ are the same and the value is 0.3. This value tells us that the test is revealing some state difference for both parents which is something we want. If we take the average, we will have a value of 0.3.

Remember that since this is how different the objects in the test are between the merge and its parents, we want to maximise this value. As such, if we maximised the average, we would be increasing the chances of survival of $t$ when $t_1$ is better because it shows small differences for both parents and not a large difference for only one of them.

If we want to maximise the state difference, and now that we have seen that taking the average is not good, we might think of just taking the maximum value between the two differences. This, however, lands us in the same situation as taking the average. For the same previous example, we would be prioritising $t$ instead of $t_1$. As such, what we want to do is take the minimum of the two differences and try to maximise it.

\[
\text{methodDistance}(t) = \sum_{m \in \text{coverMethods}} \text{distanceToLine}(m, t)
\]

\[
\text{objectDistance}(t) = \min(\text{stateDifference}(M, V_1), \text{stateDifference}(M, V_2))
\]

\[
\text{fitness}(t) = \text{methodDistance}(t) + \frac{1}{1 + \text{objectDistance}(t)}
\]

Formula 1: Fitness function formula

Since we are talking about maximising a difference, there is no limit to how different the objects between two versions can be. Because of that, we defined a distance threshold value that tells EvoSuite when the objects are sufficiently different. This distance threshold is one of the configurable properties that can be changed in UNSETTLE’s configuration file.

In the end, we have two components to our fitness function value: the distance to cover

\[
\text{methodDistance}(t) = \sum_{m \in \text{coverMethods}} \text{distanceToLine}(m, t)
\]

\[
\text{objectDistance}(t) = \min(\text{stateDifference}(M, V_1), \text{stateDifference}(M, V_2))
\]

\[
\text{fitness}(t) = \text{methodDistance}(t) + \frac{1}{1 + \text{objectDistance}(t)}
\]
each of the methods and the minimum state difference between the objects of the merge and each of its parents. The formula that defines the function is presented in Formula 1 where \( \text{coverMethods} \) is the list of methods to cover, \( M \) is the merge version, \( V_1 \) is the first parent, and \( V_2 \) is the second parent.

### 6.4 Difference Propagation

Even if we can generate tests that result in different object states between versions, this does not guarantee that the test will fail if we run it in the merge’s parents.

This, as stated by Silva et al. [21], is because of the assertions EvoSuite generates. There is no guarantee that EvoSuite will generate an assertion that tries to assert the difference in the object’s state. As pointed out by Silva et al. [21], most of the time, EvoSuite will use generic assertions. For example, it will verify the size of a list but not its contents.

In order to bypass this limitation, we could try to force EvoSuite to generate some assertion that evaluated the \( \text{hashCode} \) of each object to assess its state. However, not every class might override this method.

As a solution to this problem, we created the \text{AllFieldsCalculator} class. This class has a single method: \text{public static long allFieldsMethod(Object)}. This method transverses the various fields of the passed object using reflection and creates a unique value based on them. In its essence, it works exactly like the \( \text{hashCode} \) method.

The goal of this method is to display the state difference that was achieved during the evolutionary step. We inject a call to this method at the end of the test. If there is a thrown exception, we insert the calls right before the \text{try/catch} block.

We keep the class that contains this method in the EvoSuite code as opposed to having it in the system under test. This does not change the necessary dependencies to compile the generated tests because they always need EvoSuite for the dependencies.

Because of non-deterministic aspects of certain classes, for example from the classes in the \text{java.time} package, there are certain types that this method will not analyse. This is to make sure there is no non-deterministic behaviour in the generated test.

The same principle applies to mocked objects by the \text{Mockito} framework. The framework injects a field in the mocked objects that is only relevant to them that, because of the nature of our method, we travel through.

\(^2\text{https://site.mockito.org/} \) (accessed June 2021)
6.5 Other Changes

Instrumentation

By default, EvoSuite only instruments the target class for the test generation. Other alternatives exist, such as instrumenting the target class’ hierarchy or the classes that are called in test execution.

It is important to point out that only instrumented classes will appear in the call trace of a test or have their lines counted. As mentioned before, we use the distance to reach a method $M$ as the distance to reach any of its lines. As such, and since we might want to cover methods from classes that are not the target class, we cannot run EvoSuite with its default instrumentation.

In order to fix this, we extended the code that checks if a given class is supposed to be instrumented. If all of our target methods belong to the same class, i.e., the target class, we only instrument that one. Otherwise, if we have methods from different classes, we instrument every class.

Minimisation

After finishing the evolutionary step and before generating assertions, EvoSuite tries to minimise the test suite.

Since we want to display the state difference achieved during test generation, we do not want any of the statements removed from our test.

Before EvoSuite starts the minimisation process, we check if the criterion being used is the one for our fitness function. If it is, we tell EvoSuite to skip the minimisation step.

6.6 Summary

Given the specific nature of our problem, we extended the test generation tool EvoSuite [3] to better fit our needs.

We started by adapting the tool so that it could work with three versions of a class. Then, we extended it to receive multiple methods that a test needed to cover.

Depending on the semantic conflict we want to detect, we might have methods we do not want to appear in the test, i.e., secondary methods. To make sure these methods do not appear in the test, we marked them as synthetic during instrumentation. This way, EvoSuite will not add any calls to them in the test.

To guide the evolutionary algorithm, we created our own fitness function that optimises tests to reveal the different behaviour between the three versions of the class. This function is divided into two components. The first is related to the reachability aspect, i.e., we need to reach the method with the conflict. The second is the objects’ state difference
between the merge version and each of its parents. By maximising this difference, we increase the chances of propagating the difference between the three versions.

Since EvoSuite might not generate assertions that correctly assess the state of the objects in the test, we developed a method that works like the `hashCode` method and inject a call to it in the test for each relevant object. This way, we can assert if the state of the objects is the same between the different versions.

In the next chapter, we present the evaluation of UNSETTLE for detecting semantic conflicts and scalability. We also present the limitations of the tool and of the approach it follows.
Chapter 7

Experimental Evaluation

This work developed UNSETTLE, a tool for automatic detection of semantic merge conflicts. UNSETTLE analyses the changes performed at the AST level, compares them to a set of change patterns, and, if there is a match, tries to generate a test that reveals the conflict.

This chapter presents the goals of the experimental evaluation, the methodology used for assessing UNSETTLE and the results.

7.1 Research Questions

During this evaluation, we set out to answer the following research questions:

**RQ1**: Can we detect the different types of semantic conflicts by using automatically generated tests?

**RQ2**: Can we detect semantic conflicts present in merge instances from the related work?

**RQ3**: How does the pattern matching aspect scale?

7.2 Evaluation

The experiments presented in this chapter were executed in an ASUS VivoBook S15 laptop with an Intel(R) Core(TM) i7-8550U 1.80GHz processor and 8 GB memory.

The reported times represent the average from three executions. UNSETTLE was configured to use the default values for all of its configurable options. This means: 2 threads running in parallel when checking if a change pattern happens (one thread per pattern); a maximum time of 2 minutes when trying to match an instance to a change pattern; a distance threshold value of 0.05; and 1 minute budget for the test generator.

It is important to point out that the smaller the distance threshold value, the longer it might take UNSETTLE to generate a test that satisfies it. On the other hand, if it is a large value, the detected difference between objects might not be enough to reveal a semantic conflict.
When we say that we ran UNSETTLE with a given file, it means we inputted the three versions of that file, i.e., the base version and its variants. Also, the number of lines shown in this chapter include the comments in the source code.

### 7.2.1 Detection of different types of conflicts (RQ1)

#### Types of semantic conflicts

Despite the numerous works around merge conflicts, we did not find any data about the causes of semantic conflicts. In this way, we set out to collect the most common types of semantic conflicts in software projects.

In order to identify the types of concurrent changes in software projects that led to semantic conflicts, we performed a search through various open-source projects available on GitHub\(^1\).

For each project, we first identified its merges. For each merge that had not resulted in a textual conflict, we identified its parents. Once we had these three commits, we ran the available test suite in each one of them. If we found a test that failed in the merge but did not fail in its parents, that meant we were in the presence of a semantic conflict. A test failing in the merge when it did not fail in its parents tells us that the changes in each branch, while safe on their own, lead to the appearance of unintended behaviour when merged.

Unfortunately, this search did not identify any such situation. One possible reason for this is that the available test suite does not cover the changed parts. It might also be the case that the available test suite did reveal the semantic conflict, but since continuous integration was used, pull requests that had failing tests were not accepted.

Since our search for semantic conflicts in the wild failed, we built a catalogue of change patterns. These change patterns were inspired by mutation operators introduced by Offutt et al. [35]. We designed a total of 29 change patterns.

The change patterns produced were divided into groups. This was because the inherent change that led to the conflict could be represented in different ways (as shown in Section 4.1). Table 7.1 shows the groups for the change patterns and their description.

#### Detection of the different types

We started by creating a merge scenario for each of our change pattern groups.

Each of these scenarios contained only the bare minimum to represent the underlying semantic conflict. For example, the merge for the parallel changes to the same method contained only one class `A` with one method.

We ran UNSETTLE for each of these merge scenarios. In all of the cases, we were able to detect the change pattern and generate a test that revealed the conflict.

\(^1\)https://github.com/ (accessed June 2021)
<table>
<thead>
<tr>
<th>Group</th>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Method</td>
<td>CM</td>
<td>Update two different dependencies of a method or update one method and concurrently update one of its dependencies</td>
</tr>
<tr>
<td>Change Method and Field</td>
<td>CMF</td>
<td>Change the type of one field to a type that does not override a method while a dependency for the method is added to a method that reads the field</td>
</tr>
<tr>
<td>Dependency Based</td>
<td>DB</td>
<td>Update a method while a dependency to it is added concurrently</td>
</tr>
<tr>
<td>Field Hiding</td>
<td>FH</td>
<td>Hide the field of a superclass in a subclass and concurrently add a method in the subclass that writes the super field</td>
</tr>
<tr>
<td>Overload by Access Change</td>
<td>OAC</td>
<td>Change the visibility of an overloaded method and concurrently add a dependency to it</td>
</tr>
<tr>
<td>Overload by Addition</td>
<td>OA</td>
<td>Overload a method and concurrently add a dependency to it</td>
</tr>
<tr>
<td>Parallel Changes</td>
<td>P</td>
<td>Concurrent changes to the same entity, i.e., method (PM), constructor, (PC) or field (PF)</td>
</tr>
<tr>
<td>Remove Overriding</td>
<td>RO</td>
<td>Remove the override of a method and concurrently add a dependency to it</td>
</tr>
<tr>
<td>Unexpected Overriding</td>
<td>UO</td>
<td>Override a method in a subclass while a dependency to it is added concurrently (AO) or override an Object-inherited method and concurrently add a dependency to it (UO)</td>
</tr>
</tbody>
</table>

Table 7.1: Groups and description for the constructed change patterns.

In order to evaluate UNSETTLE in larger instances, we fabricated 10 scenarios with different types of semantic conflicts.

These fabricated merge instances were built from projects of past classes from the University of Lisbon. These are shown in Table 7.2.

The projects in question were both phases of the project from the Algorithm and Data Structures (ADS) class of 2016/2017; the project from the Introduction to Programming (IP) class of 2020/2021; the project from the Object Centred Development (OCD) class of 2011/2012; and the project from the Construction of Software Systems (CSS) class of 2020/2021.

The IP project (ID F1) revolved around the classic game “Word Search”. The first ADS project (ID F2) was the implementation of a linked list of blocks. Each block stored a finite number of elements in a circular array. The second ADS project (ID F3) was the implementation of a ternary search tree. The elements of each node were characters. The OCD project (ID F4) was the implementation of a simplified spreadsheet with cells, lines, columns, formulas, etc.. The CSS project (IDs F5-F10) was a system for the purchase of tickets for events.
Table 7.2: Fabricated merge instances and the semantic conflict present.

<table>
<thead>
<tr>
<th>ID</th>
<th>Project</th>
<th>Semantic Conflict</th>
<th>Number of Input Files</th>
<th>Number of Files in Project</th>
<th>Number of Lines of Changed Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>IP</td>
<td>DB</td>
<td>2</td>
<td>6</td>
<td>601</td>
</tr>
<tr>
<td>F2</td>
<td>ADS</td>
<td>DB</td>
<td>3</td>
<td>6</td>
<td>381</td>
</tr>
<tr>
<td>F3</td>
<td>ADS</td>
<td>DB</td>
<td>2</td>
<td>4</td>
<td>506</td>
</tr>
<tr>
<td>F4</td>
<td>OCD</td>
<td>CM</td>
<td>2</td>
<td>77</td>
<td>84</td>
</tr>
<tr>
<td>F5</td>
<td>CSS</td>
<td>FH</td>
<td>1</td>
<td>52</td>
<td>47</td>
</tr>
<tr>
<td>F6</td>
<td>CSS</td>
<td>RO</td>
<td>1</td>
<td>52</td>
<td>59</td>
</tr>
<tr>
<td>F7</td>
<td>CSS</td>
<td>AO</td>
<td>1</td>
<td>43</td>
<td>57</td>
</tr>
<tr>
<td>F8</td>
<td>CSS</td>
<td>OAC</td>
<td>2</td>
<td>43</td>
<td>369</td>
</tr>
<tr>
<td>F9</td>
<td>CSS</td>
<td>OA</td>
<td>2</td>
<td>43</td>
<td>369</td>
</tr>
<tr>
<td>F10</td>
<td>CSS</td>
<td>UO</td>
<td>2</td>
<td>43</td>
<td>202</td>
</tr>
</tbody>
</table>

Table 7.3: Results of UNSETTLE for the fabricated merge instances.

<table>
<thead>
<tr>
<th>ID</th>
<th>Change Pattern Detected</th>
<th>Semantic Conflict Detected</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Yes</td>
<td>Yes</td>
<td>447.0</td>
</tr>
<tr>
<td>F2</td>
<td>Yes</td>
<td>No</td>
<td>246.3</td>
</tr>
<tr>
<td>F3</td>
<td>Yes</td>
<td>No</td>
<td>343.5</td>
</tr>
<tr>
<td>F4</td>
<td>Yes</td>
<td>Yes</td>
<td>186.3</td>
</tr>
<tr>
<td>F5</td>
<td>Yes</td>
<td>Yes</td>
<td>16.0</td>
</tr>
<tr>
<td>F6</td>
<td>Yes</td>
<td>Yes</td>
<td>22.3</td>
</tr>
<tr>
<td>F7</td>
<td>Yes</td>
<td>Yes</td>
<td>21.0</td>
</tr>
<tr>
<td>F8</td>
<td>Yes</td>
<td>Yes</td>
<td>79.7</td>
</tr>
<tr>
<td>F9</td>
<td>Yes</td>
<td>Yes</td>
<td>82.7</td>
</tr>
<tr>
<td>F10</td>
<td>Yes</td>
<td>Yes</td>
<td>174.0</td>
</tr>
</tbody>
</table>

When testing these fabricated merge instances, we ran UNSETTLE with the modified conflicting files and tested them against all of our change patterns.

The results of UNSETTLE for our fabricated merge instances are shown in Table 7.3. As we can see, UNSETTLE was able to match the changes in the merge instances to the respective change patterns. UNSETTLE was also able to generate a test that revealed the conflict for almost every instance.

For both ADS projects (IDs F2 and F3), we could not generate a test that revealed the conflict because of the distance threshold value we used. In both cases, the return values of the methods that had the semantic conflict were boolean and there was no other difference in the objects’ states. In order to detect the conflict, we had to run UNSETTLE with a distance threshold value of 0.5.

We also created three merge instances where no semantic conflict was present and ran UNSETTLE with these. These instances were variations of the projects from the CSS class. The results are shown in Table 7.4.
Table 7.4: Results of UNSETTLE for the fabricated merge instances without conflicts.

<table>
<thead>
<tr>
<th>ID</th>
<th>Number of Input Files</th>
<th>Number of Files in Project</th>
<th>Number of Lines in Input Files</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>5</td>
<td>55</td>
<td>526</td>
<td>515.7</td>
</tr>
<tr>
<td>N2</td>
<td>2</td>
<td>55</td>
<td>258</td>
<td>272.0</td>
</tr>
<tr>
<td>N3</td>
<td>5</td>
<td>19</td>
<td>398</td>
<td>156.3</td>
</tr>
</tbody>
</table>

(a) Filtering of Sousa et al. [9]'s merge instances.  
(b) Filtering of Silva et al. [21]'s merge instances.

Figure 7.1: Filtering of state-of-the-art merge instances.

UNSETTLE did not match the changes to any of our change patterns. Since there were no semantic conflicts present, UNSETTLE did not start the test generation phase.

We performed these tests to find out if there were any bugs in our tool that caused it to report a change pattern where none existed. Since we manually constructed these instances as per our definition of semantic conflict, the only way for UNSETTLE to report a conflict would be if it had a bug.

7.2.2 Detection of semantic conflicts in merge instances (RQ2)

Merge Instances

Since our search for semantic conflicts in the wild was not successful, we collected merge instances with semantic conflicts used in related work. In particular, we used the instances identified by Sousa et al. [9] and Silva et al. [21], available in two public repositories in GitHub. The two datasets contain merge instances from open-source projects available on GitHub.

Sousa et al. [9] implemented a crawler that searched for merges where parallel modifications were made to the same method. In their experiments, regardless of the number of files in the commit, only the file with the altered method was considered.

This crawler collected 52 merge instances. Sousa et al. [9] then manually analysed these merge instances, following the process shown in Figure 7.1a. The final merge in-


\[\text{https://github.com/spgroup/mergedataset/ (accessed June 2021)}\]
stances with semantic conflicts belonged to the okhttp\(^4\) and spring-boot\(^5\) projects.

Given the small number of true positives, Sousa et al. [9] selected one of the projects they had analysed (kotlin\(^6\)) and fabricated 7 merge instances that were not semantic-conflict free and, hence, had a semantic conflict. These fabricated instances only contained the semantic conflict of parallel modifications to the same method.

Silva et al. [21] created their dataset by gathering merge instances from the state-of-the-art, including seven identified by Sousa et al. [9].

They collected a total of 38 merge instances from the state-of-the-art. Out of these, 13 had a semantic conflict as per their definition of interference (see Section 2.1.2). We identified two of them (activiti\(^7\) and swagger-maven-plugin\(^8\)) as being the result of a manual merge.

The filtering process is shown in Figure 7.1b. In the end, the merge instances with semantic conflicts were 11 and were from the storm\(^9\), antlr4\(^10\), cloud-slang\(^11\), fitnesse\(^12\), retrofit\(^13\), elasticsearch-river-mongodb\(^14\), jsoup\(^15\), netty\(^16\), and resty-gwt\(^17\) projects. The summary for the entire filtering process is shown in Figure 7.2. The two instances that we could not build were both merges from the spring-boot project.

Table 7.5 shows the collected merge instances, the dataset from which they came,

\(^4\)https://github.com/square/okhttp (accessed June 2021)
\(^6\)https://github.com/JetBrains/kotlin (accessed June 2021)
\(^7\)https://github.com/Activiti/Activiti (accessed June 2021)
\(^8\)https://github.com/kongchen/swagger-maven-plugin (accessed June 2021)
\(^9\)https://github.com/apache/storm (accessed June 2021)
\(^12\)https://github.com/Unclebob/fitnesse (accessed June 2021)
\(^13\)https://github.com/square/retrofit (accessed June 2021)
\(^15\)https://github.com/jhy/jsoup (accessed June 2021)
\(^16\)https://github.com/netty/netty (accessed June 2021)
\(^17\)https://github.com/resty-gwt/resty-gwt (accessed June 2021)
Table 7.5: Semantic conflicts present in the collected merge instances. Note that the merge instance of antlr4 contained the semantic conflict in two different classes.

the project and the change pattern present. In the Project column, elasticsearch-river-mongodb was shortened to river for the sake of readability.

Detection in Collected Merge Instances

We ran UNSETTLE for each of these 14 merge instances. We ran each of these merge instances by feeding UNSETTLE with the modified files where the semantic conflict was present and tested the instance with all the change patterns.

We also ran UNSETTLE with the seven fabricated kotlin instances created by Sousa et al. [9]. We ran each of these instances by feeding UNSETTLE with the modified file. Like the previous instances, we tested for all the change patterns.

The results of UNSETTLE for the collected merge instances are shown in Table 7.6. We were able to match all the change patterns present in the merge instances. Out of the 19 semantic conflicts present, we were able to generate a test that detected the conflict for six of them. UNSETTLE was also able to detect the change pattern in the seven kotlin instances created by Sousa et al. [9]. However, since we could not build the project, we were not able to check if we could generate a test that revealed the semantic conflict.
### Table 7.6: Results of UNSETTLE for the collected merge instances.

<table>
<thead>
<tr>
<th>ID</th>
<th>Change Patterns Matched by UNSETTLE</th>
<th>Conflict Detected by Authors</th>
<th>Conflict Detected by UNSETTLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes*</td>
</tr>
<tr>
<td>R2</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>R3</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>R4</td>
<td>Yes</td>
<td>No*</td>
<td>No*</td>
</tr>
<tr>
<td>R5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>R6</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>R7</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>R8</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>R9</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>R10</td>
<td>Yes</td>
<td>?</td>
<td>No</td>
</tr>
<tr>
<td>R11</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>R12</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>R13</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>R14</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>R15</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>R16</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>R17</td>
<td>Yes</td>
<td>?</td>
<td>Yes</td>
</tr>
<tr>
<td>R18</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>R19</td>
<td>Yes</td>
<td>?</td>
<td>No</td>
</tr>
</tbody>
</table>

Yes* – UNSETTLE had trouble generating a test that revealed the conflict but could succeed when ran with a long time budget. No* – the conflict was not detected in the original code.

**Okhttp**  
Regarding the okhttp project (R1), we were capable of generating a test that revealed the semantic conflict. However, because the semantic conflict was “hidden” behind very specific conditions, UNSETTLE had trouble generating a test that revealed the conflict.

In particular, we could only reveal the conflict if, during test generation, UNSETTLE was capable of creating a valid *url* while keeping one of the fields uninitialised.

This fact matches what Silva et al. [21] concluded from their experiments (see Section 2.2.2) where test generators need to be able to create meaningful objects to be able to reach the semantic conflicts in the code.

**Spring-boot**  
While we were capable of detecting the change pattern in both spring-boot instances (R2 and R3), we were not able to build the system in each of the merges. Because of that, we were not able to check if we could generate a test that revealed the conflict.

**Antlr**  
The antlr merge instance (R5 and R6) contained two classes with the same type of semantic conflict. We were capable of not only detecting the pattern in both but also
generate a test for each one that revealed it. An example of one of the tests generated by UNSETTLE to detect the semantic conflict in R5 is shown in Listing A.2 along with the test generated by Silva et al. [21] shown in Listing A.1. Both tests fail in the assertion that assesses the size of the set returned by the call to `getBadWords()` (lines 6 and 5, respectively).

**Storm** We were not able to generate a test that revealed the semantic conflict present in the *storm* project (R4). This is because, in order to build an object of the class that has the conflict, we need to perform a specific sequence of calls from a *builder* class. This situation shows us again that test generators need to be able to create relevant objects in order to reach the semantic conflicts.

In their work, Silva et al. [21] modified the original source code and made the private constructor accessible. This was so that EvoSuite [3] did not have to use the *builder* class to create objects. In these conditions, they were able to detect the semantic conflict. We also tried UNSETTLE with this modified version of the source code and were also able to detect the semantic conflict.

**Netty** Silva et al. [21] were not able to generate a test that revealed the PM conflict in the *netty* merge instance (R16). The reason for this was because the assertions EvoSuite generated did not inspect the state of the objects. Because of that, there was no propagation.

Thanks to our approach of assessing the state of the objects in the test (as explained in Section 6.4), we were able to detect the semantic conflict present in the merge instance. An example of one of the tests generated by UNSETTLE to detect the semantic conflict is shown in Listing A.3.

The conflict in question happened at the state level when the `decode` method was called. As such, the test starts by creating the necessary objects and then invoking the method. Since the conflict happens at the state level, the test fails in the assertion on line 20 that evaluates the state of the object that suffered the call. This shows, as stated by Silva et al. [21], that the assertions generated by test generators need to explore the objects they are evaluating.

We do not know if the tests generated by Silva et al. [21] detected the CM conflict (R17) that happened in that merge instance. Similarly, we do not know if the tests Silva et al. [21] generated revealed the CM conflict for the *retrofit* (R10) and *resty-gwt* (R19) instances.

Furthermore, we were not able to detect the other conflicts. The reason for this was that UNSETTLE was not capable of generating meaningful objects that revealed the state difference between the merge version and its parents. Once again, this verifies what Silva et al. [21] stated about the automatic generation of meaningful objects.
Table 7.7: Results of testing the matching aspect of UNSETTLE by varying the number of patterns and input files for the netty project.

### 7.2.3 Scalability (RQ3)

We selected the netty project (R16 and R17) from our dataset of real merge instances to test the scalability of the matching phase. We selected this project for two reasons. The first was that it contained more than one change pattern in the conflicting class. The second was that we were able to detect these conflicts while Silva et al. [21] could not (as explained in Section 7.2.2).

We ran UNSETTLE by varying the number of input files and the number of change patterns loaded.

We started by running UNSETTLE only with the file that had the semantic conflict, i.e., `LengthFieldBasedFrameDecoder.java`. In the next iteration, we ran UNSETTLE with two more files, keeping the `LengthFieldBasedFrameDecoder.java`. We added two other files for a final set, keeping the same previous three.

For each of these sets of input files, we ran UNSETTLE and only tested for the PM change pattern. Then, we tested for our three patterns of parallel changes to an entity, i.e., PM, PC, and PF. Finally, we tested for all the patterns.

### Matching Scalability

The results of running UNSETTLE by varying the number of input files and the number of change patterns to be tested are shown in Table 7.7. The time it took increased with the number of files that were altered. This was expected because UNSETTLE had more deltas to compute between the pairs of files introduced. More deltas also imply more possibilities when trying to match these deltas with the delta patterns which also led to an increase in the computation time.

Another reason for the increase in time from the number of files is related to the “fitness” of the change instance for a change pattern (see Section 5.2.3 for more information).
If we have more actions in a change instance, the chances of it being fit to match a change pattern increase. Because of this, UNSETTLE tried to match the change instance to more change patterns, thus increasing the computation time.

For example, let us say that we have two classes, A and B, where A suffered a parallel modification to the same method M and B is a newly created class with a dependency to M.

If we run UNSETTLE only feeding it class A, our change instance will have two deltas, both with the update method M action. Since this change instance does not have any insert class actions, UNSETTLE will not try to match it with our change pattern where a method is updated while a new dependency to it is added in a new class (shown in Listing 4.3).

Now, let us say that we run UNSETTLE by feeding it both classes (A and B). This means our change instance will have the update method M action in both deltas, and the insert class B action in one. Since this change instance has an insert class action, UNSETTLE will try to match it with our change pattern where a method is updated while a new dependency to it is added in a new class.

There was a big difference when running UNSETTLE for all change patterns with 1 or 3 files.

The fact that it did not take UNSETTLE very long to test all the change patterns with only one file comes from the previous explanation. When we increased the number of files, we increased the actions in the change instance. As such, the chances of it being fit to match more change patterns increased.

Since we were testing all of the patterns, we had change patterns that relied on the dependency relationship.

Verifying if a method $M_0$ depends on another method $M_1$ is computationally heavy. Since the change instance was fit for more change patterns, UNSETTLE tried to verify this dependency relationship when checking if there was a match. This led to the large increase in the computation time.

### 7.3 Limitations

In the evaluation process, some limitations of UNSETTLE were identified.

**Canonical Names** As mentioned in Section 5.2.1, UNSETTLE cannot work with systems that have two different classes with the same canonical/qualified name, even if these classes are in separate submodules. This limitation comes from the use of the Spoon [1] library. In order to overcome this, developers need to be more specific with the paths to the source code in the configuration file.
Classes and Enumerates During our experiences, we also discovered that the version of EvoSuite that UNSETTLE uses cannot handle enumerates when generating tests. UNSETTLE also cannot work with classes whose names are the same as the descriptors for primitive types (e.g., B, I, C, etc.). This is due to the fact that EvoSuite cannot work with such classes.

Inner Classes When testing our fabricated merge instances, specifically the ones from the ADS projects, we noticed that when the target class contained inner ones, EvoSuite could not identify the methods from the outer one. For example, we had a `ADLList` class with an inner `Node` class and, after instrumentation (which is when EvoSuite learns which methods each class has), it did not know any of the methods from the `ADLList` but knew of the ones from `Node`.

As mentioned before, we used an older version of EvoSuite and major changes and redesigns have happened since then so some of these problems might have been addressed.

Matching Time Budget The amount of time that UNSETTLE uses by default when trying to find a match of a change pattern might not be enough. Since we can configure how long we want UNSETTLE to run and can specify what change pattern to test, developers can adapt the tool’s properties depending on their situation.

For instance, developers can run it for a specific change pattern they suspect happened in the merge. Another possibility is to allow UNSETTLE to run over longer periods of time, overnight for example. If a conflict is detected, the changes would still be fresh in the developer’s mind, as opposed to detecting the conflict after deployment.

Object Generation A major limitation of the approach lies with the generation of meaningful objects. As we saw, EvoSuite and automatic test generators, in general, might have difficulty generating objects that allow the execution of the test to reach the semantic conflicts.

One possible alternative when trying to reach aspects of the code, like methods or fields, could be to use reflection. EvoSuite already has a feature implemented that allows the test to access private methods and fields. However, this does not include constructors. As such, we still face the problem of creating correct objects. As we saw, there are cases where we need to use a specific sequence of calls to create an object (as per the builder design pattern). This sequence is very difficult to achieve when we are randomly generating tests.

Executables Another limitation of the approach is that it needs executable objects, specifically during the test generation step. This is something that does not happen with static analysis as it works exclusively with the source code.
7.4 Summary

This evaluation showed us that we can use UNSETTLE to detect semantic conflicts by matching the changes to change patterns and then generating a test that reveals the conflict. However, there are a few aspects to take into account.

The first is the amount of time UNSETTLE might need when trying to check if one of the change patterns happened. As we have seen before, the time it takes for UNSETTLE to run increases with the number of files that were changed. This increase happens not only because UNSETTLE has to compute the difference between more files, but because the more files we have, the more deltas we will have that we will try to match against the pattern.

We also saw that the size of the system under tests affects the time. A bigger system means a higher number of possibilities during the matching phase. The fact we also try to match change patterns based on the dependency relationships also contributes to the increase in time as this verification is computationally heavy.

Another aspect relates to UNSETTLE’s capability to generate correct and meaningful objects. This limitation has two aspects. First, UNSETTLE needs to be able to generate objects that allow it to reach the semantic conflict. As we saw with the okhttp and storm projects, this can be difficult to achieve if we need to fulfil a specific set of conditions. Reaching the semantic conflict might not be enough to propagate it. As such, UNSETTLE also needs to be able to generate objects that cause differences to appear between the merge version and its variants. This is a difficult limitation to overcome and its directly bound to the state-of-the-art of automatic test generators.
Chapter 8

Conclusion

Semantic conflicts are concurrent changes that cause the merged result to misbehave. These conflicts are not detected during merge operations which can lead to faulty code being introduced into production. We can use tests to detect this type of conflict. However, this detection is directly tied to the quality of the test suite being used, namely whether it covers the changed parts.

In this work, we considered semantic conflicts defined in terms of emergent and lost behaviour. Emergent behaviour represents the behaviour that appears in the merged result without any of the developers writing it explicitly while lost behaviour is the new behaviour added in a branch that behaves differently in the merged result. We focused on the detection of emergent behaviour.

This work tackled the detection of semantic conflicts with automatic generation of tests guided by the information about the concurrent changes. The main idea behind this approach was to overcome the dependency on the quality of the test suite by using automatically generated tests that were guided to target common causes of semantic conflicts. The proposed solution is based on the identification of change patterns that might result in semantic conflicts and uses the information in these patterns when generating tests that will reveal the conflict.

This approach strongly relies on the ability to represent changes at a high abstraction level between two versions of a system. However, the existing change models are not appropriate to reason about semantic conflicts. They are either too generic, like the traditional line-diff model, or too detailed without the ability to represent meaningful changes, like the models used by Martinez and Monperrus [30]. Our change patterns need to address the properties of the base version and the changes performed in each branch. Because of this, we developed a new change model to represent the meaningful features of the code-entities and the actions that can be performed with them. This model specifies the state of the base version, i.e., which classes, methods, fields, etc., exist, and what concurrent changes in those entities possibly lead to semantic conflicts.

As a proof-of-concept, we developed UNSETTLE, a tool capable of detecting emer-
Chapter 8. Conclusion

gent behaviour in merges for Java code. To detect this emergent behaviour, UNSETTLE searches for a test that passes in the merge version but fails in its parents. The fact a test fails in the merge’s parents but not the merge shows that the merge has some unintended behaviour that emerged because of the concurrent changes.

UNSETTLE uses the information about the concurrent changes in the merge to guide the test generation. It starts by identifying the state of the base version and changes at the AST-level for its variants. Then, it checks this change instance (base version and deltas) against a set of change patterns. Each change pattern, if matched, has information on what class and which methods should be covered in the test.

In order to guide the test generation, we extended EvoSuite [3] to create a test that reaches the target methods and maximises the differences between the objects of the merge version and its parents.

Since the generated assertions might not evaluate the object in a helpful manner, we implemented a hashcode-like method that returns a value that is specific to each object. A call to this method is injected in the generated test so that EvoSuite can generate an assertion that evaluates this value and check the state of the object.

In the preliminary evaluation of UNSETTLE, we first used fabricated merge instances from past projects. UNSETTLE was capable of generating a test that revealed the semantic conflict in all of these instances and a test that revealed the conflict in 80% when running with the default values for its configurable properties.

We also tested UNSETTLE with 14 real merge instances collected from the related work, specifically from the work of Sousa et al. [9] and Silva et al. [21]. These merge instances contained 19 cases of semantic conflicts. We were not able to check 2 of these instances because we could not compile them. UNSETTLE was capable of generating a test that revealed the semantic conflict for 6 of these 19 cases ($\approx 32\%$).

Because of our approach to create a hashcode-like method, UNSETTLE proved useful to detect semantic conflicts that happened at the state-level of the objects. UNSETTLE is also not limited to situations where the changes were made to a single class or procedure. This is something that other approaches do not tackle as they start their approach from a situation where they know beforehand where the conflict happened.

The approach to detect semantic conflicts that UNSETTLE follows has two major limitations. The first one is the amount of time it might take to detect an instance of a semantic conflict. This depends on the number of altered files, the size of the system, and which patterns it is testing. Since we can configure how long we want UNSETTLE to run and can specify what change pattern to test, developers can adapt the tool’s properties depending on their situation.

The second one relates to the capability of test generators to automatically create objects. The created objects need to not only reach the semantic conflict but reveal the differences between the merge and its parents. Semantic conflicts can be hidden behind
very specific conditions which makes it difficult to generate meaningful objects that reach them. One possible alternative when trying to reach the conflicts could be to use reflection.

**Future Work**

We have identified a set of aspects that could improve the solution developed in this work and were left to future work.

**Refining the Change Model**  Many of our change patterns represent the same abstract change and are merely variations of one another. This is due to the lack of “or” and “and” logical operators. Having different yet similar patterns means we have a bigger number of patterns to analyse during execution, many of which share the same testing goal. Refining this model is something to be done in the future.

Related to our patterns, improving our pattern matching algorithm is also something to be explored. Instead of querying for the possible values for each variable, we could use the information of the deltas in the patterns to limit the possibilities. This way, we might improve the performance of the pattern matcher component.

**Update to EvoSuite**  As mentioned before, UNSETTLE used an older version of EvoSuite to tackle the test generation problem. Since then, newer releases have come out that probably fixed some of the problems we encountered such as enumerates and inner classes. More importantly, the tool has also suffered a major redesign. Migrating our changes to the most recent version of EvoSuite is also something to be explored in future work.

**Test Generation Verification**  Right now, UNSETTLE generates tests that cover a specific set of methods and achieve state differences. However, it does not check if the test fails in the parent versions. Improving UNSETTLE to check if a test fails in the parents before stopping the generation process is something to be developed in the future.

**Detection of Lost Behaviour**  We defined semantic conflicts with the notions of emergent and lost behaviour. UNSETTLE, however, focuses on the detection of emergent behaviour. Extending the tool to also detect lost behaviour is something to be explored in the future.

The major difference between emergent and lost behaviour is the target of the test generator. When searching for emergent behaviour, we will generate a test for the merge version. However, for lost behaviour, we need to generate tests for each of the merge parents.
Integration with a VCS  As it stands, UNSETTLE works with four versions of a system. Another feature to explore in the future would be to integrate the tool with a VCS. Instead of manually feeding UNSETTLE with the four versions, these would be automatically identified and passed to the tool during merge operations.
Appendix A

Generated Tests

Silva et al. [21]’s generated test for the antlr merge instance  The following test is one of the tests generated by Silva et al. [21] that detects the semantic conflict present in the Python2Target class of the antlr merge instance.

Listing A.1: Test generated by Silva et al. [21] that detects the semantic conflict in the antlr merge instance.

```java
@Test(timeout = 4000)
public void test0() throws Throwable {
    Python2Target python2Target0 = new Python2Target((CodeGenerator) null);
    Set<String> set0 = python2Target0.getBadWords();
    assertEquals(85, set0.size()); // (Inspector) Original Value: 85 | Regression Value: 84
}
```

UNSETTLE’s generated test for the antlr merge instance  The following test is one of the tests generated by UNSETTLE that detects the semantic conflict present in the Python2Target class of the antlr merge instance.

Listing A.2: Test generated by UNSETTLE that detects the semantic conflict in the antlr merge instance.

```java
@Test(timeout = 4000)
public void test0() throws Throwable {
    Python2Target python2Target0 = new Python2Target((CodeGenerator) null);
    Set<String> set0 = python2Target0.getBadWords();
    assertEquals(86, set0.size()); // This is the assertion that fails
    assertFalse(set0.isEmpty());
}
```

```java
Set<String> set1 = python2Target0.getBadWords();
assertNotNull(set1);
assertEquals(86, set1.size());
assertFalse(set1.isEmpty());
```
Appendix A. Generated Tests

UNSETTLE’s generated test for the netty merge instance

The following test is one of the tests generated by UNSETTLE that detects the semantic conflict present in the LengthFieldBasedFrameDecoder class of the netty merge instance.

Listing A.3: Test generated by UNSETTLE that detects the semantic conflict in the netty merge instance.

```java
@Test(timeout = 4000)
public void test0() throws Throwable {
    ByteOrder byteOrder0 = ByteOrder.nativeOrder();
    LengthFieldBasedFrameDecoder lengthFieldBasedFrameDecoder0 = new LengthFieldBasedFrameDecoder(1272, 8, 8);
    ChannelHandlerContext channelHandlerContext0 = mock(ChannelHandlerContext.class, new ViolatedAssumptionAnswer());
    Channel channel0 = mock(Channel.class, new ViolatedAssumptionAnswer());
    byte[] byteArray0 = new byte[6];
    byteArray0[0] = (byte) (-90);
    byteArray0[1] = (byte) 34;
    byteArray0[2] = (byte) 22;
    byteArray0[3] = (byte) 34;
    byteArray0[4] = (byte) 5;
```
byteArray0[5] = (byte)1;
BigEndianHeapChannelBuffer bigEndianHeapChannelBuffer0 = new
  BigEndianHeapChannelBuffer(byteArray0);
Object object0 =
  lengthFieldBasedFrameDecoder0.decode(channelHandlerContext0,
    channel0, bigEndianHeapChannelBuffer0);
assertNull(object0);

AllFieldsCalculator.allFieldsMethod(byteOrder0);
  
long long0 =
    AllFieldsCalculator.allFieldsMethod(lengthFieldBasedFrameDecoder0);
assertEquals((-2375579147664976682L), long0); // This is the
  assertion that fails

AllFieldsCalculator.allFieldsMethod(channelHandlerContext0);
AllFieldsCalculator.allFieldsMethod(channel0);
AllFieldsCalculator.allFieldsMethod(bigEndianHeapChannelBuffer0);
long long1 = AllFieldsCalculator.allFieldsMethod(object0);
assertEquals(0L, long1);
Acronyms

API Application Programming Interface. 20, 30, 42, 43, 45, 47, 49
AST Abstract Syntax Tree. xiii, 17, 19, 20, 30, 33, 42, 43, 49, 52, 63, 78
BDCI Behavioral Driven Conflict Identification. 15
CVS Concurrent Versions System. 18
DSL Domain Specific Language. 5, 29, 31, 33, 37, 38, 40, 46, 49
IDE Integrated Development Environment. 13
OO Object Oriented. 4, 13
RIP Reachability, Infection, Propagation. 54, 55
SMT Satisfiability Modulo Theories. 3, 15
TOM Testing On Merges. 14
VCS Version Control System. 1, 2, 7, 8, 13, 18, 21, 80
Bibliography


