MUTATION-DRIVEN TEST GENERATION FOR CONFLICT DETECTION IN SOFTWARE INTEGRATION

Ricardo Daniel Sequeira Wilhelm

DISSERTAÇÃO
MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software
2013
MUTATION-DRIVEN TEST GENERATION FOR
CONFLICT DETECTION IN SOFTWARE
INTEGRATION

Ricardo Daniel Sequeira Wilhelm

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Projecto orientado pelo Prof. Doutor Francisco Cipriano da Cunha Martins
e co-orientado pela Prof. Doutora Maria Antónia Bacelar da Costa Lopes

2013
Acknowledgments

I would like to express my gratitude to my project’s coordinator, Prof. Doctor Francisco Martins, for the provided support and great conversations which contributed to a joyful fulfillment of this work. It was a pleasure working with him, due to the always positive criticism and most friendly collaboration he offered, during the whole time I had him as my course teacher, and he was also one of the reasons for the choosing of this project.

I would also like to thank my co-coordinator, Prof. Doctor Maria Antónia Lopes, for the constant criticism, patience, availability and dedication until the last moment to the realization of this project.

A word of appreciation goes to my friends, especially Marco Barros, José Pedrosa, Jorge Antunes Gomes, João Alabaça Ferreira and Fernando Fernandes, for the moments of friendship, understanding and mutual assistance that were most valuable during my life in the faculty. May they live long and prosper.

Also, to my professors, who were always supportive and generally pleasant to learn from and work with, and who helped me develop my passion for computer science.

Finally, my family, for their support in the difficult moments.
To my friends, teachers, passions, and to you, reader.
Resumo

Em projetos de desenvolvimento de software, os programadores colaboram muitas vezes em equipa com o intuito de aumentar a sua produtividade. Os sistemas de controlo de versões (VCS) permitem facilitar esta colaboração, tendo a tarefa de fundir as alterações feitas por cada membro da equipa, possibilitando modificações de um ou mais ficheiros concorrentemente por vários programadores e aumentando assim a produtividade da equipa. No entanto, vários conflitos podem emergir neste processo de fusão.

Os sistemas de controlo de versões mais simples têm a capacidade de fundir linhas de texto; outros mais complexos recorrem às características sintáticas e semânticas de cada linguagem. Em resultado destas fusões podem aparecer diferentes tipos de conflitos, a maior parte detetada pelos compiladores, manifestando-se através de erros de compilação.

Entre os conflitos mais importantes estão os conflitos semânticos, i.e., situações em que é possível fazer a fusão ao nível textual, mas o comportamento do programa passa a não ser correto. Estes conflitos são, por certos autores, classificados em duas categorias: conflitos estáticos e comportamentais. Os conflitos estáticos podem ser detetados a partir da análise do código fonte do programa, por exemplo durante a análise semântica do compilador. Os conflitos comportamentais afetam o comportamento dos programas, que após a fusão pode já não corresponder ao que cada membro esperava do código que produziu. No contexto de linguagens orientadas a objetos (OO) como o Java, este tipo de conflitos surge normalmente devido ao uso de primitivas específicas do OO como a herança, a redefinição e a sobrecarga de métodos.

Este documento descreve uma técnica desenvolvida para detetar conflitos comportamentais em projetos Java, assim como um protótipo de uma ferramenta que concretiza parcialmente esta técnica. O objetivo da ferramenta, denominada MC-MODS (Merging Conflict and Mutation Operation Detection System), é diagnosticar classes importantes de conflitos em Java. A convicção é que ao reportar os conflitos detetados no contexto de VCSs irá ajudar a forma como os programadores trabalham cooperativamente.

A técnica desenvolvida explora a semelhança que existe entre mudanças que ocorrem em resultado de um processo de fusão e as mutações efetuadas no contexto do teste de mutação, uma técnica que consiste em introduzir alterações em programas (chamadas mutações) para desenhar novos testes ou avaliar a qualidade dos testes existentes.
As alterações efetuadas por cada membro da equipa podem ser vistas como mutações—mudanças ao código do membro sempre que chama o processo de atualização. O MC-MODS, uma vez integrado com um sistema de controlo de versões, permitirá analisar as mutações produzidas por diferentes membros da equipa, procurando métodos que possam ter sido afetados por estas, ou seja, cuja semântica, ou comportamento, possa ter mudado. No que se segue, estes são chamados métodos alvo. A ferramenta MC-MODS foi implementada sob a forma de um plug-in para o IDE Eclipse, pois esta é uma das plataformas de desenvolvimento Java mais usadas e suporta um grande número de sistemas de controlo de versões.

A técnica de deteção de conflitos desenvolvida assenta ainda na definição de um catálogo com várias classes de mutações típicas que provocam alterações ao comportamento de programas em Java, causadas pelos processos de fusão de alterações dos sistemas de controlo de versões. As operações de mutação presentes no catálogo cobrem diferentes primitivas OO, nomeadamente herança e sobrecarga de métodos, ocultação de atributos e mudança de acesso (i.e., mudança de uma classe de pacote ou mudança do modificador de acesso de alguma classe ou membro). Cada operação no catálogo contém um ou dois destes referidos aspetos. A ferramenta é também capaz de detetar mudanças do tipo de retorno ou do corpo de métodos, bem como classes ou membros adicionados ou removidos.

Para cada operação de mutação foi criado um caso de teste que instancia a operação, composto por três versões do código fonte, necessárias para detetar o conflito presente na versão final. Estas três versões são constituídas por: (a) uma versão base, i.e., uma versão não alterada antes do processo de atualização, só depois do qual um programador deve fazer alterações; (b) uma versão denominada original, i.e., uma versão que contém as alterações do utilizador da ferramenta e do VCS e (c) uma versão merged (fundida) que representa o programa depois do processo de fusão que contém as alterações do utilizador e dos restantes membros da equipa. Estas alterações são obtidas a partir do repositório remoto que o VCS usa.

Resumidamente, o funcionamento do MC-MODS é o seguinte: as três versões referidas acima são comparadas entre si com o objetivo de detetar os métodos alvo. Os métodos alvo são calculados através da análise da árvore de sintaxe abstrata de cada classe e pela procura dos métodos e atributos que possam ter tido a sua semântica alterada, determinando se os métodos alvo chamam (no caso de serem métodos/construtores) ou usam (no caso de serem atributos) de facto algum desses membros relacionados. São então gerados automaticamente testes JUnit para averiguar se a semântica dos métodos alvo calculados terá sido afetada. Estes testes envolvem 1) a criação de duas instâncias da classe a que o método pertence, cada uma na sua versão (original e merged), 2) a invocação do método alvo sobre ambos os objetos criados e 3) a comparação do estado desses objetos no final da invocação e dos seus resultados (se aplicável). As duas instâncias são criadas
com classloaders especializados, sendo a ponte entre estes estabelecida através da ferramenta Transloader (esta ferramenta permite usar um objeto construído num classloader diferente do classloader corrente).

Espera-se que as operações de mutação do catálogo definido no contexto deste trabalho cubram grande parte das características orientadas a objetos em programas Java que mais frequentemente causam conflitos comportamentais e que a técnica desenvolvida ajude efetivamente a diagnosticar, tão cedo quanto possível, conflitos que ocorrem em resultado do trabalho cooperativo neste contexto.

Uma das características importantes da solução apresentada é não serem reportados falsos positivos. Já a existência de falsos negativos está intimamente relacionada com a cobertura dos testes que são gerados. Apesar de no MC-MODS se ter optado por uma versão minimalista no que diz respeito à geração de testes, outras hipóteses são possíveis, sendo que será sempre preciso estabelecer compromissos relativamente ao custo acrescido que uma maior cobertura necessariamente exige.

Este trabalho foi realizado no contexto da disciplina de Projeto de Engenharia Informática do Mestrado em Engenharia Informática da Faculdade de Ciências da Universidade de Lisboa.

Palavras-chave: testes, mutação, fusão, conflito orientado a objetos, integração contínua.
Summary

Programmers often collaborate together in order to increase their productivity in software development. Version control systems intend to facilitate this collaboration, by automatically integrating the changes of each member, so they can easily work on a single software artifact at the same time. When such a system integrates these changes, by merging them, sometimes it causes conflicts—unexpected errors caused by the merge process—simply because before the merge each member expected his code to have a certain behavior, but after the merge this no longer happens. They may be syntactic errors—like a keyword in the wrong place—or semantic errors—like a wrong value returned by a method.

This document describes a technique for detecting important classes of conflicts in Java and a prototype of a tool that partially implements this technique. The tool was implemented in the form of a plug-in for the Eclipse IDE, i.e., as an add-on for a widely used platform for developing software that is known for its extensibility, multi-language capability and support of several version control systems. The plug-in, called MC-MODS (Merging Conflict and Mutation Operation Detection System), aims to detect behavioral merge conflicts for Java projects. This is a type of semantic conflicts that is not detected by static analysis techniques, which arises when programs show an unexpected behavior after the merge process. MC-MODS automatically generates tests designed to detect these conflicts with a high probability that are inspired by a technique known as mutation testing. The proposed technique is based on a catalog of mutation operators capturing the main source for semantic conflicts in the context of Java projects, involving object-oriented features like overriding and overloading. Once integrated with a version control system, MC-MODS will support the early detection of many important semantic conflicts in Java and hence can improve the way Java developers work collaboratively.

This work was done in the context of the course Projecto de Engenharia Informática belonging to the Master Course in Computer Science of the Faculty of Science of the University of Lisbon.

Keywords: version control system, merge, conflict, object-oriented, mutation, testing, continuous integration.
# Contents

## List of Figures

## List of Tables

## 1 Introduction

1.1 Motivation ........................................... 1
1.2 Contributions ...................................... 4
1.3 Structure of the document ............................ 4

## 2 Background and Related Work

2.1 Version Control Systems - Background .......................... 5
2.2 Merging and Conflict Detection .............................. 6
2.3 Types of Conflicts ..................................... 6
2.4 Conflict Detection Approaches .............................. 7
2.5 Object-oriented Features ................................ 9
2.6 Faults and Mutation Operators ............................ 12

## 3 Conflicts and Detection Technique

3.1 Merging as Mutation .................................. 15
3.2 Behavioral Conflicts in Mutations ......................... 17
3.3 A Catalog of Behavioral Merge Conflicts .................. 18

## 4 Design and Implementation

4.1 Overview ............................................. 33
4.2 Design by Component .................................. 36
   4.2.1 MainHandler .................................... 37
   4.2.2 Target Calculator ............................... 37
   4.2.3 Code Generator ................................ 40
   4.2.4 Test Runner .................................... 40
   4.2.5 Target Invocator ............................... 41
   4.2.6 Object Comparator .............................. 42
4.3 Detailed Design ..................................... 42
List of Figures

2.1 An example of a yo-yo graph. ........................................... 11
3.1 Merging and mutation operations. .................................. 15
3.2 Unexpected Overloading - Case 1. ................................. 16
3.3 Unexpected Overloading - Case 2. ................................. 19
3.4 Unexpected Overloading - Case 2 - Inverse. ................... 20
3.5 Unexpected Overriding 1. ............................................. 21
3.6 Unexpected Overriding 1 - Inverse. ............................... 22
3.7 Unexpected Overriding - Case 2. ................................. 23
3.8 Attribute Hiding. ....................................................... 24
3.9 Access Change 1 - Code. .......................................... 25
3.10 Access Change 1. ..................................................... 26
3.11 Access Change 2 - Code. .......................................... 26
3.12 Access Change 2. ..................................................... 27
3.13 Access Change 2 - Inverse. ...................................... 28
3.14 Access Change 1 - Reverse. ..................................... 29
3.15 Access Change 2 - Reverse. .................................... 30
3.16 Access Change 2 - Reversed Inverse. ......................... 31
4.1 A VCS time-line. ...................................................... 34
4.2 MC-MODS: Inputs. ................................................... 35
4.3 MC-MODS: Data Flow Diagram - Part 1. ....................... 37
4.4 MC-MODS: Data Flow Diagram - Part 2. ....................... 37
4.5 MC-MODS: Control Flow Diagram - Part 1. .................. 38
4.6 MC-MODS: Control Flow Diagram - Part 2. .................. 38
4.7 MC-MODS: Class Diagram. ....................................... 43
List of Tables

2.1 Relation between faults and mutation operators. . . . . . . . . . . . . . . . 12
Chapter 1

Introduction

This report is an integral part of the curricular unit “Projeto de Engenharia Informática (PEI)”, course of the second year of the “Mestrado em Engenharia Informática” of the Informatics Department in the Faculty of Sciences of the University of Lisbon, with the duration of nine months.

1.1 Motivation

Programmers rarely work alone, as they often belong to teams of multiple people, who work together to create programs in a collaborative way, in order to increase the quality of the produced software, the productivity of the members and the success of the team. Software integration is hence an important concept in software development. It consists in combining several closely related variants of a system, each one incorporating changes made by each of the development team members, and the joining of all these variants, also called branches or versions, into a base version that represents their consolidated content. This practice provides the teams with control over the changes of each member, approving or rolling them back if necessary.

Several systems are used to improve the coordination between the teams, which become even more complicated if the people involved in these projects are distributed across different time zones and geographical locations. Version Control Systems (VCS) are one kind of tools that support this coordination, and employ mechanisms that ensure that each version of every software artifact can be controlled, that is, with each new development introduced by a team member, e.g., new or altered functionality, a new version (also called revision) of the artifact is created, and the modifications undergo merge algorithms that blend them into single artifacts. These artifacts can be items such as files or documents.

Commercially available version control systems such as CVS\footnote{http://cvs.nongnu.org/}, Git\footnote{http://git-scm.com} and Mercurial\footnote{http://mercurial.selenic.com} use text-based merging, which is simple but also of limited use. The most common ap-
proach is the representation of lines of text as atomic units, known as line-based merging. With this kind of merging, the text lines that were inserted, modified, deleted or moved, can be detected in concurrent modifications, as well as common lines, but it has the disadvantage of not handling two parallel changes in the same line very well, because only one of these can be chosen and they cannot be combined.

The limitations of text-based merging are even more severe when applied to programs; everything is treated as a piece of text and, hence, the specific semantics of software artifacts are not taken into account [10]. In particular, text-based merging gives no guarantees that the program will behave as expected.

A conflict emerges when the version control system is unable to integrate parallel changes automatically. A user must typically resolve them by combining the changes manually, or discarding the ones he does not want.

As text-based merge techniques only detect very basic conflicts, different approaches have been taken, in more modern merge tools, to detect the more complex types of conflicts. On the one hand, there are approaches based on syntactic merging, that make use of the syntax in software artifacts. For example, syntactic merging ignores code comments, line breaks and tabs, used to improve the readability of code. This is typically supported by a more sophisticated representation of programs, such as (parse) trees or graphs. On the other hand, approaches based on semantic merging also take into account some semantic issues. This type of approaches can often detect occurring conflicts that syntactic merge tools cannot, such as undeclared variable errors or incorrect number of arguments in procedures. These are called static semantic conflicts, which are detected trivially, as most compilers will signal a problem. Nevertheless, static semantic merging is still insufficient, as parallel changes can give rise to unexpected behavior, which reflects itself in the execution flow of the program and the relation between its inputs and outputs. This brings us to the second type of semantic conflicts, behavioral conflicts, that is the most relevant type of conflicts to this thesis: a program’s behavior (recognized by its externally visible operations/outputs), after concurrent changes were safely merged, is different from what was expected.

The early detection of conflicts is very important in software development, because the sooner these are detected, the easier they are to resolve, as the ideas are still fresh in the programmers’ minds and less effort is spent on remembering and ultimately figuring out what was done. Some conflict detection tools adopt a pessimistic approach when dealing with dubious situations (i.e., when they are not able to confirm whether there is a conflict or not) and report false positives — warnings that a conflict exists where it does not. A different pessimistic approach is to prevent the parallel editing of one artifact by more than one developer by locking. This is inadequate in most cases and obviously lowers the usefulness of the tools, as it does not support parallel development of single artifacts. Most version control systems adopt a optimistic approach, in which each developer can
work on a personal copy of a software artifact, but it has the price of introducing the need for merging, where each parallel change is integrated into base versions, which can be consisted of single software artifacts [10].

It is quite important and difficult to find the right balance between the precision and recall in conflict detection, and different approaches have been considered. One approach is to detect conflicts before new changes are checked in the repository, which has the advantage of reporting them to the developers as soon as they arise, at the cost of producing false positives or conflicts that disappear after check-in. Another approach is to check for conflicts after one has checked in/commit new code to a repository, and has the advantage of reducing the number of false positives (conflicts that are detected but do not exist at check-in time) at the cost of detecting them at a later phase.

This work addresses the detection of behavioral semantic conflicts in Java, i.e., in the context of object-oriented programming. The aim is to contribute with a technique that can improve the effectiveness of state-of-the-art version control systems in conflict detection. Based on the idea that tests are key to discovering semantic conflicts [5], the generation of tests for conflict detection will be explored, taking inspiration in mutation testing.

Object-oriented programming (OO) languages possess complex primitives, inherent to OO, such as inheritance, polymorphism, encapsulation and dynamic binding, which give rise to complex types of conflicts during parallel development. For instance, if programmers do not consider notifications of changes by other team members, do not take the structure of the used classes in consideration or do not even know the internals of the used classes (due to proprietary implementation and private source code), they might be using inheritance and overriding without even knowing it. Calling a merge operation after a concurrent change can be seen as a mutation operation, as both are direct changes in the code or text itself, from the point of view of one developer. If the mutations contain these object-oriented features, it is likely they will contain anomalies.

The last aspect this work is related to is mutation testing. Mutation testing is a technique that helps software testers evaluate the quality of software tests. It is based on the production of mutants: small syntactic changes introduced into the programs under test, for example changing a plus (+) operator to a minus (-). This concept, also known as fault seeding, is usually used to measure and improve the quality of test suites [11, 14]. Modifying a program’s source code or byte code in this way improves the detection of bugs and the quality of test suites, as the suite is considered defective if it does not detect and reject the (normally faulty) mutated code. Therefore it allows the detection of weaknesses and the development of more effective tests. A test case that distinguishes the program from its mutant is considered effective at finding the faults in the program, or more commonly referred to as killing the mutants.

Calling a merge operation after a concurrent change can be regarded as a mutation:
both are direct changes in the code itself that might result in failures or conflicts. So, conflict detection can be achieved through the generation of tests that kill mutants with high probability.

### 1.2 Contributions

This work focuses on the detection of behavioral semantic conflicts caused by merge operations in Java programs, which are undetectable by state-of-the-art static analysis tools. Furthermore, this project contains the following contributions:

- a diagnosis of important classes of conflicts in Java programs, by introducing a novel technique based on mutation testing. Reporting these conflicts in the context of version control systems can improve the way developers work collaboratively.

- a design and an implementation of a plug-in for the Eclipse IDE that applies this conflict detection technique and presents conflicts to developers in order to help them identify problems.

### 1.3 Structure of the document

The remaining of this research is organized in the following way:

- Chapter 2 presents some publications that are relevant to this work, and introduces important concepts.

- Chapter 3 delineates the central ideas of this work, and a catalog of mutation operators is presented.

- Chapter 4 describes the design decisions and the implementation details of MC-MODS.

- Chapter 5 contains the evaluation of the implemented tool and its results, and delineates the critical analysis and possibilities of future work.
Chapter 2

Background and Related Work

This chapter reviews some of the research literature regarding software merging and the detection of conflicts that arise in this process. It overviews different classifications for conflicts, detection approaches as well as object-oriented faults and their associated mutation operators.

2.1 Version Control Systems - Background

As previously mentioned in the introduction, a version control system is a piece of software that intends to help project development teams create better programs by controlling each change made by each team member. These systems provide control over a set of data, also called a repository, i.e., a collection of many individual items, such as files or documents. A version, or revision, represents each important modification, or change.

When a team member wants to publish his changes to the other members, he must make a commit or check-in so his modifications are integrated into the system so they become final and available to all the users. An uncommitted copy of data is called a 'working copy', and it is stored locally on the developer’s computer. The commit and the check-in, also called “push” in Git\(^1\) are normally two separate steps, but some version control systems do not differentiate them, and the commit phase is not even present in one of the most popular ones, CVS\(^2\). Changes are normally forwarded to the local working copy, and afterwards pushed to the remote repository as necessary, but in this work I will refer to “commit” as both the commit and the push.

Afterwards, if a different team member wants to check-out these changes, he makes an update, which synchronizes changes made from the repository into his local working copy.

Every time a user makes an update, the version control system attempts to merge the changes of the repository with the ones in the working copy, e.g., if each user writes

\(^1\)http://git-scm.com
\(^2\)http://cvs.nongnu.org/
one line of text, both of the lines will be present in the merged version, but in more complicated examples, sometimes the merge process is not successful. This happens because the system cannot make the merges automatically, and this creates a conflict. In the more simple text-based approach, one such example would be if the users modify the same line of text. The different types of conflicts are described later in this chapter.

2.2 Merging and Conflict Detection

As most version control systems adopt an optimistic approach, which represents the degree of permission of developers to edit software artifacts in parallel, they also need a merging process to integrate the changes of each participant into a new shared version, and possible emergent conflicts need to be resolved.

There is an important distinction between merge tools, based on their representation of software artifacts: textual, syntactic, structural, and semantic. A better alternative than text-based merging tools, that represent software in flat text files, is the usage of more structured forms of artifacts, e.g., a parse tree, or to consider semantic information.

Some types of conflicts can be detected through compilation, static or dynamic analysis tools. Static analysis is performed without executing programs, and is usually done by reviewing a version of the programs’ source or object code, highlighting possible coding errors or proving their mathematical properties. Dynamic analysis is based on the execution of programs, analyzing their inputs, outputs and behavior, e.g., using software testing techniques.

Also important in conflict detection is the concept of awareness. It helps to detect conflicts early in a way in which the members report to the co-workers each change they made, in regard to files, types, or program elements. However, this may cause developers to be overloaded with notifications of changes that are irrelevant to their tasks, and it also requires investigation each time a notification could cause a conflict, and these could, for this reason, be ultimately ignored, which is counter-productive. This is even more difficult if the program elements have complex semantic dependencies like polymorphism and dynamic binding.

2.3 Types of Conflicts

Most approaches to conflict detection distinguish two types of more general conflicts: direct and indirect. Direct conflicts may occur when two versions of the same artifact are changed concurrently, for example when two or more developers alter the same file at the same time, knowingly or not. Indirect conflicts are caused by a change in two or more different artifacts. In both cases the conflicts emerge because the artifacts contain incompatible, parallel changes across them. Awareness techniques only help to detect
direct conflicts, and are unable to support indirect ones.

In [3], Guimarães and Rito Silva propose the more fine-grained classification of conflicts than those specified in the introduction, as follows:

- **Structural conflicts**, created during the background merging of the authors’ software tool, are mostly naming conflicts of folders and files represented by nodes and attributes in the evolution graph/tree of each team’s project.

- **Language conflicts** consist of pieces of programs that are syntactically invalid. In mainstream VCS, the compiler of the language in question detects and reports these conflicts each time the merged system is updated.

- **Test conflicts** are related to running automated tests in the merged system, which may result in an execution flow that reached methods changed by different team members, using the Reflection API. These methods could have been deleted or even never existed.

- **Behavioral conflicts**, one type of **semantic conflicts**, are the last and the most relevant type of conflicts to this work. A program’s behavior, recognized by its externally visible operations/outputs, after it was merged with textual-based techniques, may be different from what is expected.

The other type of semantic conflicts, static semantic conflicts, referred to by Mens [10], are detected by the compiler, e.g., an invalid number of arguments in a method’s call or an undeclared variable error, so their relevance is lowered.

### 2.4 Conflict Detection Approaches

Tom Mens in [10] discussed some alternatives for merge techniques, each one with its own conflict detection algorithms:

- two-way or three-way merging;
- textual, structural, syntactic, semantic, or operation-based merging;
- state-based or change-based merging;
- reuse versus evolution.

The detection of syntactic conflicts is typically based on the representation of programs as trees or graphs, each one with its own capabilities. Graph-based merge approaches also have the capability of detecting semantic conflicts, using def-use relations, which are the explicit links between the definitions and invocations of procedures and are
made explicit in this type of approach. In tree-based approaches there is no such capability, as the def-use relations are not made explicit in a parse or abstract syntax tree, but this structure can be populated with additional information, as in context-sensitive merging \[18\], where the abstract syntax tree is populated with context-sensitive relationships which express these def-uses.

Other interesting conflict detection techniques analyzed in \[10\] are merge matrices, conflict sets and some semantic conflict detection techniques. A *merge matrix*, also called a conflict table, makes it possible to detect merge conflicts by performing a simple table lookup, in a table which defines merge functions and allows the definition of merge policies. *Conflict sets* group together potentially conflicting combinations of operations based on the semantics provided by an application, containing kinds of operations and associated merge conflicts that can differ dramatically.

Detecting all semantic conflicts between different versions of programs is generally undecidable \[6, 10\]. This problem can be eventually overcome by narrowing down the merge algorithm to a well-defined domain (e.g., a type of application). However, a domain-independent approach is in general required. The cost is payed in terms of the accuracy, and solutions are typically based on approximative techniques that try to detect as many conflicts as possible without sacrificing efficiency, instead of detecting all possible conflicts.

**Conflict Patterns.** According to Offutt et al. in \[3\], one way of detecting behavioral conflicts is by searching for *conflict patterns*: logical conjunctions of facts about the program’s elements and their semantic dependencies in the merged system that identify potentially unwanted behavior.

**Reuse Contracts.** Tom Mens in \[9\] presented a formal and domain-independent approach to detect structural and behavioral inconsistencies in a uniform way, when merging parallel developments of software artifacts. By using *reuse contracts*, defined by a set of method descriptions each one consisting of a unique name and optionally an *abstract* annotation or a specialization clause, potential behavioral conflicts can be detected in a very straightforward way, by documenting each modification explicitly, and verifying if certain conflicting combinations of operations are made.

**Self Testing Code.** Self Testing Code is a strategy that may significantly help dealing with semantic conflicts. It involves the practice of writing comprehensive automated tests together with the functional software, i.e., the code. In test-driven development, developers actually write tests before the functional code, but there are many advantages writing them after rather than before, and merely the existence of these tests is important, not how they were created. Tests also do not help with resolving the conflict once it is discovered,
but its detection is a “big part of the battle”.

**Before check-in.** Sarma and van der Hoek [16] implemented a workspace awareness tool, Palantír, that is intended to eliminate isolation in workspaces, informing each developer of ongoing parallel changes that other developers make. It has the capability of accepting early human intervention should a potential conflict arise. Palantír improved conflict detection and resolution and decreased the number of conflicts in frequently checked-in code.

Guimarães and Rito Silva [3, 6] developed a continuous integration tool that merges each developer change in the background in order to facilitate automatic detection of conflicts. It has the advantage of not interrupting programming flow like manual detection, and reported the conflicts in an IDE, giving the possibility of early resolution while the changes are still fresh.

**After check-in.** O’Reilly et al. [15] extended CVS, a revision control system, with Night Watch, that offered automatic monitoring and notification facilities to report to members of a team changes that were relevant to them. It supported independent development and communication, as everyone was meant to be kept informed of relevant activity by other team members.

Brun et al. [2] proposed a stand-alone conflict detection tool, Crystal, that aimed to reduce the number of false positives (detected conflicts that in fact do not exist), using speculative analysis. This concept consists in anticipating and suggesting the actions of developers and executing them in the background, to help developers identify, manage and prevent conflicts, detecting these in check-in time.

One lightweight approach, *Semantic Diff* [7], makes use of local dependence graphs for behavioral conflict detection. Given two versions of a procedure, the tool generates a report summarizing the semantic differences and the effects of the modification, based on approximations to their input-output behavior.

### 2.5 Object-oriented Features

Object-oriented aspects of programming, namely inheritance, polymorphism, encapsulation and dynamic binding, bring additional expressiveness to programming languages, but also new anomalies and fault types [13]. Similarly, these aspects have a huge impact on the potential conflicts that may arise in parallel development. Hence, in the next paragraphs, I present an overview of these concepts in the context of Java, the programming language that will be the focus of this work.

*Encapsulation* allows objects to restrict access to their members, attributes and methods, by other objects. There are four distinct access levels supported by Java, represented
by access modifiers: private, protected, public and default (also called package). A private member is visible only by the class in which it is defined. If the access level is not specified, it defaults to package, which allows visibility to classes in the same package, but not subclasses in other classes. A protected member is visible by the class itself, subclasses and classes in the same package, and a public member is available to any class in any inheritance hierarchy or package, as long as its class is also public.

In Java, the change of access modifiers can have an effect on static and dynamic binding, so its potential for both expressiveness and faults is further increased, because it can change the meaning of a program [17]. The compiler binds callable methods, for example, and by changing their accessibility, others can or cannot be called instead, and this may or may not cause a compilation error. If it is not detected by the compiler, it is a task for this proposed technique.

Every class has exactly one immediate parent (or “superclass”), as Java does not support multiple class inheritance. A subclass inherits its parent’s and his ancestors’ members, and it either can either use them as defined, override the methods, hide the member attributes or add new members. They can also use the parent’s members using the keyword “super”, e.g., super.someMethod().

Contracts are used by software designers to provide formal and precise interface specifications for components, like methods. These can be, in the context of the Java language, informally described for instance in the Javadoc documentation or be formally defined in a contract language such as JML[3]. A contract consists of two parts: the requirements (pre-conditions) upon the client of the method, and the promises (post-conditions) made by the provider. When methods of a (super)class are inherited by a subclass, the methods’ contracts are inherited as well.

Method overriding occurs when a method defined in a child class has the same signature, i.e., name, arguments and a compatible result type, in regard to a method in a superclass. It allows subclasses to redefine inherited methods, using a different implementation. In order to honor the contract, the contract of the overwritten method must not require more or promise less, i.e., the pre-conditions must not be stronger or the post-conditions weaker.

Variable hiding occurs when defining a variable in a subclass with the same name of an accessible inherited one, which has the effect of hiding the inherited variable from the child class, even if the types are different, so it cannot be normally accessed, except for example using the keyword super.

Polymorphism is achieved when two or more objects of different classes define their own unique behaviors and yet share some of the same functionality of the parent class. Each object has a declared type — the type in the declaration statement, such as Parent p; , and an actual type, which can be from any compatible type with the parent, and

---

is defined through instantiation statements (e.g., \( p = \text{new Child();} \)) or assignment statements (\( p = p\text{Old}. \)). This concept is closely related with dynamic binding, which occurs when the compiler is not able to resolve method calls or determine the actual type of variables. These bindings can only be done at runtime.

**Overloading** is defined by the use of the same name for methods in the two classes, and a different set of arguments/parameters. This can be a cause of semantic merge conflicts, as one method can be unexpectedly called instead of another. The compiler is responsible for choosing the right specific version for a particular circumstance, and more specific methods have priority. It needs to find a method having a parameter type which is a subclass of the parameter types of all other overloading methods. Overloading is easily confused with overriding that occurs between a class and one of its subclasses, but overloading can occur between two methods in the same class or in different classes. The latter case can happen when a method is inherited in a subclass that contains a method with the same name but a different parameter type.

Additionally, member variables and methods can belong to the class rather than individual objects. Members that are associated to the class are called *class or static variables/methods*.

Polymorphism and dynamic binding can complicate the understating of interactions in programs. Sometimes it is hard to understand which methods are, or even can, be executed. To illustrate that the execution can sometimes “bounce” up and down among levels of inheritance has been called the yo-yo effect [13]. The Figure 2.1 shows an example of “yo-yo graph" which contains all possible actual executions in the presence of dynamic binding.

![Figure 2.1: An example of a yo-yo graph.](image)
2.6 Faults and Mutation Operators

One of the hardest problems of developing object-oriented software is visualizing the often complex interactions that occur when using the aforementioned aspects [1]. Offutt et al. [13] collected some fault types that include the use of inheritance and polymorphism. These class-level faults are language-independent, but the language used affects how they manifest themselves.

The authors of Inter-Class Mutation Operators for Java [8] also established a relation between these faults and their correspondent mutation operators, showed in Table 2.1. Some faults require multiple mutation operators, and also some of the operators cover more than one fault.

In the same paper, Offutt provided a list of behavioral mutants, which modify the behavior of programs, and structural mutants, which modify their structure. In a more recent paper of the same author, [14], some operators were merged, some were divided, and others were introduced, namely PCI, PCD and PCC, which are relevant and related to type conversion that, in conjunction with inheritance, polymorphism and dynamic binding, allows the behavior of an object to change with different actual types, and so they have been added to the table. They are related with the fault “Inconsistent Type Use”, because type cast instructions change the actual type of variables.

<table>
<thead>
<tr>
<th>Faults</th>
<th>Class Mutation Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inconsistent type use (context swapping)</td>
<td>PNC, PRV, PCI, PCD, PCC</td>
</tr>
<tr>
<td>State visibility anomaly (possible post-condition violation)</td>
<td>IOP</td>
</tr>
<tr>
<td>State definition inconsistency</td>
<td>IHD, IHI</td>
</tr>
<tr>
<td>State definition anomaly</td>
<td>IOD</td>
</tr>
<tr>
<td>Indirect inconsistent state definition</td>
<td>IOD</td>
</tr>
<tr>
<td>Anomalous construction behavior</td>
<td>IOR, IPC, PNC</td>
</tr>
<tr>
<td>Incomplete construction</td>
<td>JID, JDC</td>
</tr>
<tr>
<td>Overloading methods misuse</td>
<td>OMD, OAO, OAN</td>
</tr>
<tr>
<td>Access modifier misuse</td>
<td>AMC</td>
</tr>
<tr>
<td>static modifier misuse</td>
<td>JSC</td>
</tr>
<tr>
<td>Incorrect overloading methods implementation</td>
<td>OMR</td>
</tr>
<tr>
<td>super keyword misuse</td>
<td>ISK</td>
</tr>
<tr>
<td>this keyword misuse</td>
<td>JTD</td>
</tr>
<tr>
<td>Faults from common programming mistakes</td>
<td>EOA, EOC, EAM, EMM</td>
</tr>
</tbody>
</table>

Table 2.1: Relation between faults and mutation operators.

The faults presented in this table are some of the causes for conflicts in object-oriented programs. Some of these include the aforementioned object-oriented features of programming languages, like overloading and attribute hiding, and are closely related to them. For example, it is possible to misuse or implement overloading methods incorrectly, define
states inconsistently or construct objects incompletely, due to using those features incor-
rectly. Also interesting is the misuse of important keywords like static, super and
this, that may cause an incorrect behavior in programs, because they cause bindings of
variables to be changed, for example one uses the keyword super to refer to something
not of the current class, but of the superclass.
Chapter 3

Conflicts and Detection Technique

In this chapter, I start by explaining two central ideas: merging operations can be compared to mutation operations, and the need of tests to discern which mutation operations can be behavioral conflicts. These ideas are explained in detail in regard to an example of a typical behavioral conflict: unexpected overloading. Then, I present a catalog of additional conflicts that have been identified. This catalog was used as a base for the iterative process for the collection of mutation operations that capture these conflicts.

3.1 Merging as Mutation

In this section, I discuss how merging can be regarded as a process of mutation and how different types of behavioral conflicts that can arise during mutation can be considered mutation operations. These ideas are discussed using a simple example of two developers working over the same repository.

Figure 3.1: Merging and mutation operations.

In Figure 3.1 a branch and a merge operation in a version control system are repres-
sent. When a developer wants to work on his project independently, he makes a check-
out of the head—the most recent commit—and has thereafter retrieved a local working
copy in which he can make the changes. The checkout operation is also called a branch,
because he is creating his separate working copy to work on. All the diagrams in the
catalog follow this representation.

On the left side, two developers, be it Garfield and Richard (identified with the green
color and red color respectively), made branches at the head, making changes simultane-
ously, and then committed their changes, merging the results into a version that consoli-
dates all of their modifications.

On the right side of Figure 3.1 the merge operation is represented in the form of a
mutation operation, in the point of view of Garfield. We can see that the branch was made
by him, and then a mutation in his code occurred, a small change made by Richard, and
it was integrated in the merge operation. The mutation operations in the diagrams are
always presented in the point of view of Garfield, that is, he is the developer who always
makes his changes first, and Richard makes the mutation in Garfield’s code.

In Figure 3.2 we can see an example of a conflict caused by a merge operation, re-
lated to overloading, which is complex and therefore can have an unexpected presence in
merged programs.

Suppose that Garfield and Richard made a checkout, that is, they imported the work-
ing copy of the Shape application source code into their mainstream VCS. The appli-
cation functionality revolves around geometry, containing shapes and facilities to create
and move these shapes. In this case we can see that it contains a single class, named
Shape, which has a method move, designed to move a shape to a location defined with two Number coordinates. This means that, for example, when move(3, 5) is invoked, it will move the shape to location (3, 5). Number is an abstract class of the java.lang package, superclass of classes like Integer and Double which wrap primitive number values of type int and double respectively.

Then, Garfield decides to make a method that resets a shape’s location, by transposing it to the origin (moving it to location (0,0)), and writes a method, reset(), which calls move with both parameters having the value 0. He makes this change and commits it in the end. Richard, at the same time, decides to create a method move with parameters of type int, i.e., move(int dx, int dy), that moves a shape along a vector (x, y). Because both developers wrote their methods at the same time, none of them knowing the intentions of the other.

We can see that Richard’s mutation, method move(int dx, int dy), will be included in the merged version, along with Garfield’s method, reset(), so we conclude that the version control system merges Garfield’s code and Richard’s mutation.

### 3.2 Behavioral Conflicts in Mutations

The other central idea is that, as already mentioned, mutation operations in version control systems can originate behavioral conflicts. Next, I present a technique to detect these conflicts, i.e., that is able to identify mutations resulting from merging, in Java.

Each branch/local working copy is correct from the point of view of the developer that owns it. However, this may not be the case in the merged result, as there is the possibility of conflicts, which arise when the two parts are integrated together. In particular, the expectations of one of the developers regarding his changed programs may happen not to hold in the merged program.

The technique’s idea is to determine if the versions made by each developer are semantically equivalent, i.e., the code behaves as expected by each developer who produced it, after the merge operation. If not, this corresponds to a conflict which must be reported. For this purpose, two versions must be compared: the one before and the one after the merge operation.

First, the technique achieves this by deducing if these merge operations do, or do not, correspond to mutation operations, by checking if they include the object-oriented features presented in the catalog. This is done by verifying if every class member is related to any other one in terms of object-oriented relationships, each one containing two members.

Next, methods must be calculated that call or use these members (methods and constructors are called, fields are used). In most cases of the behavioral conflicts captured in the following catalog, a method is expected to call or use a class member in this relation-
ship, but in the merge operation calls or uses the other one. This change in a method’s binding is called a *binding change*.

After this, the presence of conflicts is discerned by automatically generating tests that, based on target methods, i.e., methods that may have been affected by the mutations and which semantics have changed, create instances of the classes that own the methods. One object of each version is created, and the target method is then invoked on each object. The object states and invocation results, if applicable, are compared next, and if they are not equal, we are in the presence of a conflict. This happens because the two versions compared are not semantically equivalent, i.e., they have a different behavior and have produced different results.

Going back to Figure 3.2, we can see the manifestation of a conflict, related to overloading. Class `Shape` in the merge result contains both methods, the one with instances of the (more general) superclass `Number` as parameters, and the one with instances of the primitive type `int`, wrapped in the (more specific) subclass `Integer`. It is worth noticing that this conflict depends on the point of view of these two developers. Each one is only aware of the code he himself wrote and has an expectation regarding the behavior of the code he had worked on.

In this case, a conflict manifests itself in the merged version of the two concurrent changes. Richard’s expectations are not violated but Garfield expects that the result of calling `reset()`, for a given `Shape` object, moves the shape to the origin, but instead it leaves the shape in the same place, as the call to `move(0,0)` in `reset()` will be bound to Richard’s `move(int,int)` method.

This happens because calling `move` with two `int` parameters will force the compiler to choose Richard’s `move` that also has `int` parameters, because it is more specific.

### 3.3 A Catalog of Behavioral Merge Conflicts

In this section, I present a catalog of behavioral merge conflicts. The code in each class is colored according to the developer that made the change. I will use the notation `reset() ⇒ move(0,0)` to represent that a method `reset()` calls another method `move` with the value 0 in both parameters, `R[h]` to say that a member accesses (the value of) the attribute `h` and `R,W[h]` means that a member accesses and changes (reads and writes) the value of the attribute. Attributes are also called `class fields`.

Note that each conflict has its own inverse situation. For example, in the case of overloading we have a conflict that is the inverse of unexpected overloading, in which we start an overloading situation as one developer removes a method with more specific parameter types, like `int`, while the other introduces a call that will be unexpectedly bound to the method with more general types of parameters, such as `Number`. 
Unexpected overloading - Case 2. This case also represents an unexpected overloading, but introduces a new concept: **accessibility**. This is referred to **encapsulation** in section 2.3. Recall that a private member cannot be called, i.e., is inaccessible, from outside the class it belongs to.

In Figure 3.3, Garfield adds method `reset()` in class `Square`, and commits. At the same time, Richard changes the accessibility of method `move(int x, int y)` from private to public, updates his working copy with Garfield’s changes at the head, calls a merge operation to merge his and Garfield’s changes, and commits. The merge operation was clean, because both developers changed different files, however, a merge conflict exists in the final head state.

The reason for this conflict is that Richard changed `move(int x, int y)` from private to public, so it became accessible by Garfield’s method `reset()`, forcing it to be called by `reset()`, and so it has an unexpected behavior in Garfield’s point of view.

Figure 3.4 represents the inverse case of this mutation operator, where the method `move(int, int)` is changed from public to private. This also causes an unex-
pected behavior, as the method \texttt{reset()} suffered a binding change in this case as well, this time in the opposite direction. The more specific method \texttt{move(int x, int y)} stopped being accessible from class \texttt{Square} (private members only are visible to members in the same class, not subclasses), so forcing it to be bound to method \texttt{reset()}.

Figure 3.4: Unexpected Overloading - Case 2 - Inverse.

\textbf{Unexpected overriding - Case 1.} In Figure 3.5, Garfield changes class \texttt{Square}, subclass of \texttt{Shape}, again by adding method \texttt{Square.reset()}. At the same time, Richard changes the same class by adding method \texttt{move(int dx, int dy)} to move squares by some distance relative to their current location, i.e., adding the given values to the current coordinates of the shape, and commits. He updates his working copy with Garfield’s changes, calling a merge operation, and commits. The merge operation was clean, because both developers changed different files, however, a merge conflict exists in the final head state.

In this case, when Garfield invokes the method \texttt{reset()} on a \texttt{Square}, it will move the \texttt{Shape} by the distance \((dx, dy)\) from its current location, instead of moving to location \((x, y)\), as expected for all \texttt{Shapes}. The reason is that calling \texttt{reset()} on a
Square will then call `Square.move()`, not `Shape.move()`, as the compiler will choose `Square.move()` because it is in the same class as `reset()`. Also, it is only possible to invoke `reset()` on a `Square`, because class `Shape` does not have this method.

Figure 3.6 represents the inverse case of this mutation operator, where the method `Square.move()` is removed by Richard, instead of being added. This causes an unexpected behavior, as the method `reset()` suffered a binding change in this case as well, this time in the opposite direction.

Method overriding, just as method overloading, is important for testers to ensure that the correct method is invoked.
Unexpected Overriding - Case 2. In some object-oriented languages, e.g., Java and C#, constructor calls to polymorphic methods also execute the method that is closest to the instance type that is created. Figure 3.7 describes a potential anomaly in construction behavior of objects, which also relates to overriding. The class Shape is now extended by class Triangle. Also, Shape has a method resize() that resizes a shape, and class Triangle containing an attribute $h$ which represents a triangle’s hypotenuse, the longest side of a triangle.

Garfield decides to include a call to the method resize in the constructor of class Shape. That means when a Shape is created, it will automatically be resized to a predetermined size. At the same time, Richard decides to override method resize() by creating method Triangle.resize(), which modifies a triangle’s size—the length of its sides, including $h$. Both developers make a clean merge after committing, and the conflict manifests at the end.

That is, for class Triangle, the closest version of resize() is Triangle.resize(), which means that when a Triangle is being constructed, the call made to resize() in Shape’s constructor (called implicitly by class Triangle)
actually executes \texttt{Triangle.resize()} instead of \texttt{Shape.resize()}. This results in a data flow anomaly because of attribute \(h\). Because of the order of the construction, the state space of class \texttt{Triangle} (its attributes) will not have been constructed, i.e., \(h\) will not have been initialized, and the assumptions or pre-conditions of \texttt{Triangle.resize()} have not been satisfied prior to construction (e.g., \(h \neq \text{null}\)).

\textbf{Attribute hiding.} Attributes, or state variables, are members of a class that can also be inherited by subclasses. If a local variable is introduced in a class definition and if it has the same name as the inherited attribute, this one is hidden from the scope of the subclass (unless it is explicitly qualified, as in \texttt{super.v}, related to the fault “super keyword misuse” and the mutant ISK in Table 2.1), so references to this attribute will refer to the attribute of the subclass. Although this is not a problem if all inherited methods of a class are overridden, but commonly this is not the case, as typically there will be at least one inherited method that is not overridden. In this case, there is the possibility of a data flow anomaly if a method that normally defines an inherited variable is overridden in a subclass where this variable is hidden by a local definition.
In Figure 3.8 we can see the nature of this conflict. Richard added the attribute `size` to the class `Square`, so the state variable hides the inherited one of the superclass (it may or may not have been his intention). The method `resize` added by Richard will in the merged version now refer to `Square.size`, and not `Shape.size`.

![Figure 3.8: Attribute Hiding.](image-url)
Access change. Access modifiers are a part of one powerful aspect of Java, *information hiding*, which in turn is related to *encapsulation*, a design principle that intends to protect parts of the source code and data from being accessed by other code in other parts of the code. They have been described in Section 2.5. Refactoring affects the modularization of the program, and it does not alter the externally visible behavior of programs, but changing access modifiers in Java can have an effect on static and dynamic binding, which represent the semantic of programs.

Moving a class without regarding their accessibility can cause compilation errors. For example, moving a class \( B \) with a default (also called 'package') modifier to another package produces a compilation error if other classes use \( B \) and are not in the same package. This problem is therefore resolved trivially and it is not a conflict, but there are complex and relevant situations which involve accessibility.

![Figure 3.9: Access Change 1 - Code.](image)

For example, when the class \( B \) shown in Figures 3.9 and 3.10 is moved to another package, the program’s meaning is changed, as the method \( m \) in class \( A \) will now, instead of calling \( m(\text{String}) \) in class \( B \), call \( m(\text{Object}) \), as the former method is no longer accessible by \( A \) (it has the default accessibility: "package" or "package-protected") and because \( \text{String} \) is more specific than \( \text{Object} \), in terms of the \( \text{String} "\text{abc}" \). This change of meaning can be detected by observing a change in the static binding of the method call, given by the compiler, but this is not always sufficient, as we can see in the following example.

The following case creates one further type of conflict, combining the aspects of accessibility and overriding, the latter being the case where a class extends another. As we can see in Figures 3.11 and 3.12, Garfield changed the accessibility of method \( m \) in class \( B \) from *public* to *default*. Meanwhile, Richard moved this same class from package \( a \) to package \( b \). They both commit and make a merge, and in the merged result, a conflict manifests itself. Moving class \( B \) to another package changes this program’s meaning because of the change in binding, due to overriding: \( m(\text{String}) \) in class \( A \) is no longer being overridden, and calling \( m(\text{String}) \) on \( A \) no longer dispatches to the implementation in \( B \), but to the one in \( A \), due to the type casting at the beginning of the instruction, and \( B \)’s methods no longer being accessible as they were moved to package \( b \).
Next, the inverse situations of these cases are shown. In relation to access change, there are two relevant types of inverses. We can reverse the modification of the modifier, i.e., change a method from private to public (denominated inverse, see Figure 3.13), as well as move a class from an external package to the same package of the class with the calling method, and vice-versa (denominated reverse, see Figures 3.14 and 3.15). The cases where both of these modifications happen have been called reversed inverses (see Figure 3.16).

Figure 3.10 represents the same previous case, but with the changed modifier reversed,
i.e., it was changed from \texttt{default} to \texttt{public}.

The conflicts presented above can all be simply resolved by adapting the access modifiers correspondingly, but only if they are detected, and their detection in real programs is quite difficult.

Some situations have not been presented here because they do not cause conflicts. For example, the inverse of \ref{5.10}, i.e., the switching of method \texttt{m(String s)} by Garfield from \texttt{default} to \texttt{public} does not cause a binding change, as this same method, in both the original and the merged version, is called by \texttt{n()}. Its reverse case is of course also invalid.
Figure 3.13: Access Change 2 - Inverse.
Figure 3.14: Access Change 1 - Reverse.
Figure 3.15: Access Change 2 - Reverse.
Figure 3.16: Access Change 2 - Reversed Inverse.
Chapter 4

Design and Implementation

One of the tasks of this work consisted in the design and development of a prototype for a plug-in for the Eclipse IDE that, according to what was described in section 3, has the objective of detecting behavioral Java conflicts, that emerge from concurrent development and are not detectable by static analysis techniques. The tool, MC-MODS (Merging Conflict and Mutation Operation Detection System), achieves this by calculating the changes made by other team members, deducing which mutation operations captured in the catalog are reflected in these changes. At the same time it produces automatically generated tests that detect, with high probability, the mutants associated with these operations.

MC-MODS was implemented as a plug-in for the Eclipse IDE, due to the purpose of the tool—its future integration with a version control system—and because these systems are popularly used with IDEs. These ‘Integrated Development Environments’ consist of platforms where programmers develop software, with the help various tools, such as compilers and interpreters, and a user interface. Version control systems are, as already mentioned, also included in the more popular IDEs, and they further enhance software integration, and ultimately human productivity.

This chapter contains a description of the design and implementation of this plug-in, its components and packages, as well as a description of the main classes of each package and their functionality. It also provides a detailed run of the tool, containing an example of a generated test class and a reported conflict, in light of the base example, unexpected overloading.

4.1 Overview

A plug-in, for the Eclipse IDE, consists of a component that provides a service in the context of the Eclipse workbench, and is a contribution of a set of new tools to extend the functionality of the IDE. These pluggable components can be configured into a system at its deployment time, and can also extend other existing plug-ins, therefore supporting the extensibility of the Eclipse platform.
MC-MODS is meant to be integrated with a version control system. During the update process of the system, i.e., after a user updates his working copy with the changes in a remote repository, these changes and his local changes are merged. After this, possible conflicts may emerge—this is the intended moment to use this plug-in.

In Figure 4.1, we can see an example of a time-line of a project, developed using a version control system. Suppose that Garfield, at some point in time, made a check-out from the remote repository of the VCS and began working. Let this version be called base. After Garfield has made some changes, he commits them to the repository. Let this version he has now be called original. Richard, another team member, did the same, at some other point in time, and committed his changes before Garfield. Let this version be called remote, as these changes came from the remote repository, from the point of view of Garfield. This code will contain mutations to Garfield’s code—possible semantic differences that MC-MODS is designed to detect. When one of the members calls an update process, these changes will be consolidated into a merged version.

![Figure 4.1: A VCS time-line.](image)

Therefore, for each Java project, this plug-in receives these three versions as input: (a) the base version—an unaltered version of the program, prior to the update; (b) the original version—the changes of the user running this plug-in, plus the ones in the base version; and (c) a merged version which represents the program after the merge process and contains both changes: the ones from the user and the ones received from the remote repository. These three folders represent the input of the plug-in, as shown in Figure 4.2.

We have no need for the remote version because it is not easily obtainable from a VCS, in contrast to the others, and we are supplied with the merged version that contains everything we need, as by comparing the members of the base version and those of the merged version, we can find out which members were added or deleted by Richard. This comparison is done first between the base and the original versions to find out Garfield’s changes, then between the original and the merged version to find out Richard’s changes.
The integration of MC-MODS to one such system has not currently been made, therefore I made the assumption that the folders with these versions are already present, as well as a file containing their folder names.

The objective of MC-MODS is to see if these versions, committed by each member, are semantically equivalent to each other, as this is the source of behavioral conflicts. It achieves this by detecting any mutation operations captured in the catalog present in these changes. After this, the tool calculates target methods, which consist in the methods that have been affected by the mutations, i.e., methods which semantics have changed due to the merge process. Then it automatically generates tests that create two objects of each version, invoke the target methods on these objects, and compare the object states and the method results, if applicable. If the compared artifacts are not equal, the versions are not semantically equivalent and it corresponds to a conflict. Each test is designed to check if the given objects to compare have equal states, so for each failed test, a conflict is reported.

Next I describe how these tests are calculated. The plug-in starts by detecting the changes of each of the three aforementioned folders, first between the ‘base’ and the ‘original’, then between the ‘original’ and the ‘merged’ versions. Each of these sets of changes is called a change set. The first change set is the one we are normally interested in, i.e., the one that contains Garfield’s changes. Then, every method in this set is checked against every other member in the ‘original’ version, to see if it is in a relationship in terms of mutation operators (e.g., if it overloads another member). The plug-in then generates a call graph which shows members that call at least one member in relationship, also decorated with field uses, and filters out irrelevant members. What remains are ‘target’ methods, which then tests are generated for. After that, classes that contain tests are generated, which are designed to create two objects, of the ‘original’ version and of the ‘merged’ version, invoke the target member on each side, and determine if there is a conflict or not, comparing the object states and the method results, if applicable.
4.2 Design by Component

The functionality of MC-MODS is distributed through six components:

- MainHandler is responsible for controlling the execution flow of the other components,
- Target Calculator calculates the target methods by using the detected mutation operation and a call graph,
- Test Generator generates the code for the tests,
- Test Runner compiles and runs the generated tests,
- Target Invocator creates the necessary objects to compare, and
- Object Comparator makes the comparison of the objects and method results.

The following Figures show the layout of the project and its components. In Figures 4.3 and 4.4 we can see the inputs and outputs of each component, and the interconnections between the components, representing the data flow diagram of the project. Figure 4.3 shows that the three source folders are passed to the MainHandler component, which passes the control to Target Calculator, supplying this component with the folders' names. Target Calculator then reads each of the folders and calculates the change sets, mutation operators and the call graph, and deduces the target methods which are passed to the Code Generator. This component then generates the classes with the tests, which are used in one go by the next component, Test Runner. The component Test Runner then compiles and runs each test in sequence, using JUnit. The next component, Target Invocator, uses Transloader, a component external to this project, to create both objects (by cloning objects with different classloaders than the ones the objects belong to), invokes the target methods, and calls Object Comparator to compare them and/or the method invocation results, as seen in Figure 4.4 to determine the presence or absence of conflicts.

We have seen how the data is transmitted between the components. But the control flow is a bit different. In Figure 4.5 we can see that the ChangeSet Calculator delegates the Code Generator to generate the classes with the tests. After they are generated, the ChangeSet Calculator compiles the tests, and fires the Test Runner to run them, using JUnit.

While the tests are being run (see Figure 4.6), the Test Runner component calls the Object Comparator, which in turn uses the Transloader to clone the objects with the different classloaders.
4.2.1 MainHandler

MainHandler is the main component of MC-MODS, called right after the main button is pushed, and is responsible for receiving the control of the plug-in and controls the execution of the other components as needed, as well as storing the location of the plug-in and the workspace of the user.

4.2.2 Target Calculator

The Target Calculator component is responsible for calculating the target methods that the tests are designed to invoke. Its functionality consists in (a) processing each Java file of each version, creating folders if necessary, (b) generating a parser and an abstract syntax tree (AST) of each compilation unit created, (c) deducing the changes between the
three versions of the project, first between the ‘base’ and the ‘original’, and then between the ‘original’ and the ‘merged’ version; (d) calculating the mutation operators from these change sets, and (e) generating a call graph that will contain the target members. This plug-in’s representation of Abstract Syntax Tree (AST) is, in simple terms, a list of class descriptions, each of these containing the signature of the class in question, the list of the classes’ members, and another class description representing the superclass. Each compilation unit has an AST assigned to it, and it contains a type declaration (excluding internal classes), which is used to create one class description. This component also contains a custom visitor for the ASTs, navigating the Java element tree and collecting the classes’ fields, methods, constructors, as well as field uses and method calls inside methods’ bodies, package declarations and internal classes of each compilation unit. The interesting aspect about the AST functionality of MC-MODS is that it is capable of detecting and binding method invocation and field access instructions. For example, it is able to detect
a field call in a method’s body by binding it to an existing field in the current class or even in a different one. The ASTs are generated via the Eclipse AST tool, which belongs to the Eclipse JDT (Java Development Tools[1]).

Next I will describe how the change sets are calculated. Given two lists of class descriptions, that correspond to one of the three versions, they are compared in the following way: For every class in the first list, it is compared to the corresponding class in the second list. A class and its corresponding class have the same signature, and the equals method is used to compare them. If the corresponding class cannot be found, it is assumed it was deleted (from the first version to the second), and a new change, an instance of ClassDelete, is added to the result. Otherwise, they are internally compared. If there are members that do not exist in the first list, but do exist in the second, it is deduced that they were added. If they do exist in the first, but not in the second, they were deleted. This goes for all three types of members: fields, methods and constructors. The next phase is done using the second list as reference, i.e., for every class in the second list, it is compared to the corresponding class in the first list. The other two differences are: instead of ClassDelete, an instance of ClassAdd can be created, meaning that a new class was added in the second version; and another check for internal changes is not necessary. After these are calculated, the change set is filtered, for example, if a class or a member was deleted in one version, and a similar one was added in another, they are inspected to see what is different. MC-MODS is able to detect changes in modifiers, package names and method return types and bodies. Both of the changes involved (the add and the delete) are replaced with an instance of Change, which contains the software artifact of the first version and the changed data of the second. This collection, i.e., the change set, is then sorted to ease readability and debug.

Because of the complex cases captured in the catalog, and AccessChange being of a different type than for example Overloading, the need to unite the change sets (the “base->original” and the “original->merged”) surfaced, so from here on, “change set” will refer to the union of these two.

Given this unified change set, the mutation operators are calculated in the following way: for each member $m$ found in the change set, the classes (class descriptions) of its source (which can be either base, original or merged) are checked to see if $m$ is related to any of the classes’ members in terms of mutation operators, captured in the catalog. MC-MODS currently detects attribute hiding, method or constructor overloading, method overriding and return type changes. Superclasses are checked as well (to check for overriding members). The call graph is calculated in the following manner: for each ClassDescription in the original version and each member found in the change set, it checks if one calls or uses the other, or vice-versa. Recall that methods and constructors are called, and fields are used. A member can call or use another member, directly

or indirectly, the latter case being when e.g., if \( a \) calls \( b \), and \( b \) calls \( c \), then \( a \) calls \( c \). This is called a transitive closure. As some changes are related to classes and because classes are not affected by the current mutation operations of the catalog, class changes are ignored here. When this component is verifying if a member calls another, it begins by finding each one. If it does not find one of the members, or the first member is a field, it returns false. Then, it checks the member’s method and field calls, to make the transitive closure. If the member has no such calls, the method returns false, if the member is already visited, it returns true.

The call graph is stored in a HashMap of instances of the class JavaMemberSig. A HashMap is like a normal List, that stores entries, each one composed of a key and a value. In these data structures, each value can be found resorting to a hash table, that maps each key to its value using a hash function. The JavaMemberSig class, like the name implies, represents a Java member’s signature. A member, in Java, is either a field, method or constructor.

### 4.2.3 Code Generator

Code Generator is the component responsible for generating the Java code for the comparison tests. It initializes the paths to the original and remote source folders, opens the Java source file for writing and generates the imports, the package name, the class name, the test method, annotated with @Test according to the conventions of JUnit, and the rest necessary for the class. It then generates the parameters of the target method, and the target method itself, containing its package and class names, modifiers, and in the case it is not a constructor, its return type and method name (it also contains a reference to its parameters). After that, it initializes the classloaders of the ‘original’ and the ‘remote’ versions, with the mentioned paths at the beginning of the paragraph, as well as the list that will contain the conflict reports, forthcoming in [4.5] and finally the test method, of the infrastructure, that will make the object comparison. At the end, the method call to print the conflicts, and the JUnit assertion are generated. The test class example in [4.5] illustrates all of this.

### 4.2.4 Test Runner

The Test Runner component is designed to compile and run the generated tests. It first begins by finding an installation of the Java Development Kit (JDK)—which contains the needed Java compiler—as well as preparing the classpaths with the .jar files that the compiler needs, and calling the compilation tasks. The .jar files that MC-MODS depends on and includes are: JDT Core v3.8.3, JUnit v4.10 and Transloader v0.4. After this, JUnit is called to run the classes that were compiled.

One interesting part of the implementation is used when the compiler cannot be found.
If this is the case, the component tries to find it by going directly to the Java installation folder, located in the $\text{java.home}$ property, and importing the compiler from the folder, beginning with "jdk", contained herein. Another interesting and challenging part was adding each .jar to the classpath. This and the previous aspect were implemented using filename filters, which return the files, or folders, that match a certain criterion. In the first case, it must be a folder with a name starting with “jdk”, in the second, it must be anything that ends with “.jar”. Probably the most challenging part of this component was creating the compilation tasks and the test run tasks, and initializing them the required classpaths, which cannot be changed after run time and must contain each important .jar.

### 4.2.5 Target Invocator

Target Invocator is the component called when JUnit runs the test classes. After verifying if the target is a method or a constructor, it loads the target’s classes, from each version, using the classloaders provided in each test. Then it attempts to retrieve a constructor from the class of the original version, using predefined rules, and generates its parameters, to create the first object. The second object, belonging to the merged version, is cloned from the first, via the Transloader, using the classloader of the original version. Next, the target methods are loaded, from each class, using reflection, their parameters are generated, and they are invoked. After the next component makes the comparison, the found conflicts are added to the conflict list.

I will now define one important concept which API was mentioned briefly in the related work and that was widely used in the plug-in’s implementation: reflection. Reflection is a computer program’s ability to examine and alter the structure and behavior of an object at runtime. Particularly in Java, it allows applications to perform operations that would otherwise be impossible. In the case of MC-MODS, I have used reflection for accessing the private fields, getting methods and constructors directly from a class—using their names and/or parameters—and getting classes using their name.

In case of the Target Invocator component, using reflection, the classes of the target methods can be acquired directly, so the constructors can be used to create instances of these classes.

The target methods are found in the following way: First, MC-MODS finds each of the method parameters’ classes. Primitive types are treated specially, because their classes cannot be found via reflection, so they are retrieved from a HashMap which contains all primitive type classes (e.g., the class $\text{Integer}$ is found normally, but not type $\text{int}$). The found classes are stored into an array, which is simply handed, with the method’s name, to the method of the $\text{java.lang.reflect}$ package which retrieves the method, which is also searched in superclasses.

The Transloader interface is used to wrap objects referencing classes from potentially foreign classloaders. In the case of MC-MODS, it starts by cloning the first ob-
Object into the second by first wrapping the first object, creating an instance of the class ObjectWrapper, which is cloned to create the second object, which is also wrapped.

When invoking the methods, the first is invoked directly using reflection, the second is invoked from the second wrapped object. In this moment, Transloader finds the corresponding method in the second class automatically by receiving an instance of the class transloader.InvocationDescription, which describes a method invocation by method name and parameters.

The constructors are retrieved in the following manner: the list of every constructor in the class is retrieved using reflection, and sorted thereafter. MC-MODS contains a class, ConstructorComparator, which sorts constructors alphabetically and by number of parameters in ascending order. In most cases, the parameters of a constructor (or method) are not primitive types, but objects, so this retrieval method is recursive, and supports circular dependencies.

When generating the parameters for the constructors and the target methods, the parameter types are checked if they correspond to primitive types, which are then randomly generated using the java.util.Random methods. MC-MODS contains a method to generate Strings randomly. If the parameters are not primitive, parameters are generated again for the constructors of the classes.

### 4.2.6 Object Comparator

Object Comparator is the component that makes the comparison of two objects, first by comparing their classes, or values if they are primitive types, an array comparison is made if they are arrays, and finally, if all these methods are inconclusive, their class attributes are compared. These class attributes, or fields, are obtained via reflection. If the objects’ classes have superclasses, their fields are considered as well. One interesting aspect is that attributes can also be classes, so the implementation of this component is recursive, and supports circularities (e.g., a circular list of objects).

### 4.3 Detailed Design

MC-MODS is organized into three main packages, as shown in Figure 4.7.

The main package contains the classes that belong to every Eclipse plug-in: an Activator and a handler, called MainHandler, which are the first ones to be run upon the start of the plug-in.

The Activator class controls the plug-in’s life cycle, allows accessing and initializing its preferences, and it is loaded initially when it starts. Besides the Activator, a plug-in needs handlers, which are implementations of behaviors for commands and can be represented in options in Eclipse’s dropdown menus, or buttons in the IDE’s toolbars. This
plug-in has one button, associated with a command, which calls the class that implements its behavior—MainHandler. In the future, when MC-MODS is integrated with a version control system, there will be no more use for a button: the conflict detection mechanisms will start automatically with the commands of the VCS.

The tests package contains classes meant for testing purposes, i.e., the ones used to test the good operation of the project and its main functions, such as object comparison with circularity, the correct collection of the classes’ elements and classes in different packages. Each of the classes in this package extends junit.framework.TestCase, a class that belongs to the unit testing framework that is JUnit, and which defines the fixture to run multiple tests.

The infrastructure package contains the infrastructure of MC-MODS and the following packages.

The changes package contains every change that this plug-in detects in Java projects. Abstract classes represent more general changes, including a Change itself, and changes to classes, fields, methods, and constructors. The more concrete classes, which extend these abstract classes, represent added, deleted or changed members, as well as changed modifiers, package declarations, return types, and method bodies.

The main package is the main package of the infrastructure, containing some of its most important classes. The CodeGen class is the class responsible for generating the tests that determine the presence of conflicts. The Compare class serves for the comparison of any two objects. The Template class invokes the target methods and generates their parameters. The Locator class has utility methods to find method signatures in their containing data structures, and print found conflicts.

The mutops package contains each type of mutation operator in the catalog:

- AccessChangeModifier
• AccessChangePackage

• AttributeHiding

• Overloading

• Overriding

• the abstract class MutationOperator, superclass of all these, and

• FlexibleOperator.

The FlexibleOperator class is capable of storing a class-related or a member-related mutation operator. Its subclasses are AccessChangeModifier and AccessChangePackage, as they can affect both classes and members.

The others package contains the following classes:

• CompareException—the class representing a conflict. A throwable exception signifying that the detection of a conflict was successful.

• ConstructorComparator—a utility class to compare constructors in order to facilitate sorting of data structures with constructors.

• PrettyPrinterTree—another utility class that generates an abstract syntax tree of the current compilation unit and prints each node.

• RootClassLoader—this is the custom classloader used to load the two different classes from each version. It loads them based on their location and a root name, which is normally their package name. If the class to be loaded does not start with the root name, the default class loader is called, otherwise, a new class is defined. Two instances are created for each test.

• Rules—this class was initially used to store the rules to determine if two given members are affected by the object-oriented features present in the catalog, like overriding and overloading. However, these have been moved to the signature classes themselves, so it only contains modifier-related methods.

• SyntaxTree—this is the class that generates the abstract syntax tree mentioned before. A much more complex version of the PrettyPrinterTree, both extending org.eclipse.jdt.core.dom.ASTVisitor, generates abstract syntax trees from each compilation unit found, visiting each node, and collects their members, internal classes, method bodies, parameters and class packages.
The package `signatures` contains the aforementioned data structures used to represent member and parameter signatures, and classes found in the abstract syntax tree. The class `ClassDescription`'s objective is to store all the information of a class, as in its members, name, packages, internal classes and superclasses. From only one instance of this class, it is possible to retrieve all the superclasses lying above it in its hierarchy. A project, i.e., one of the three ‘versions’, is represented by a list of class descriptions.

### 4.4 Implementation

This section describes some details about the implementation of the conflict detection mechanisms designed to process the cases of the catalog.

The conflict detection and reporting for all the provided mutation operators in the catalog was successful.

Next I will describe some aspects and tools that were explored during the implementation of MC-MODS and that have contributed to its increased duration.

Until the late part of the implementation, the Code Generator and Target Invocator components received the change set and mutation operators, and it had to be changed, as the code (of the generated tests) only needs the target methods.

Also at the referred phase, the calculation of “affected” methods (later renamed as “target” methods) was done in the Target Invocator component, using the change set and the mutation operator in question. As all test cases are executed in the same way (the change set and mutation operators are not relevant for the code generation), this functionality had to be moved to the Target Calculator component.

Before deciding that the implemented tool was a plug-in, different approaches were tried to compile and run the tests. The use of symbolic links, special types of files that contain references to another file or directory, in a way related to a normal shortcut, behave as if operating on the target file directly. Apache Ant\(^2\) has been explored as well to facilitate this task. It is a Java library and command-line tool that puts processes in motion, using build files and targets with their own dependencies, widely used for building Java applications. However, the objective was to compile and run tests from the Eclipse IDE, like in some version control systems, and a plug-in for this platform was deemed more adequate.

The use of the Mockito\(^3\) mocking framework has been considered, because it was thought it would be helpful to generate call graphs for fields, to see which fields are called by which methods. But the objective of this framework is to provide stubbing and verifications for behaviors, and the production of call graphs was easier, with the AST of the Eclipse JDT.

---

\(^2\)ant.apache.org

\(^3\)http://refcardz.dzone.com/refcardz/mockito
A tool that was also thought useful for the generation of the call graph was Soot\footnote{\url{http://www.sable.mcgill.ca/soot/}} a Java optimization framework to analyze and transform Java bytecode. However, the use of this tool was unintuitive, and it was quite heavy, in relation to dependencies, functionality and documentation one had to dig through to find call graph functionality, so it was concluded that it was unusable for a Eclipse plug-in and an unnecessary dependency.

The first tests, before the code generator was implemented, were first designed resorting to reflection, but this too proved inadequate due to the presence of more than one classloader, which causes conflicts when two classes with the same name are loaded, which was one of the central obstacles that were overcome. The solution was the implementation of signatures, classes that represent Java member signatures, such as \texttt{JavaMemberSig}.

Other light implemented features were a file that deletes all the generated code, and the support for comments in the files containing the paths to each file in each input folder.

### 4.5 A System Run

This section provides a step-by-step tool run for the overloading example already considered in Section 3. I describe the functionality of the tool in detail, after receiving this example conflict case, beginning from the source folders, the generation of the test in the middle, and the object state comparison at the end. The corresponding reported conflict is also displayed, making part of the output of MC-MODS.

Next, a system run is described, representing our base example, Overloading (see Figure 3.2 on page 16), more specifically the \texttt{reset} method. The objective is to determine if the found operator caused by an addition of the method \texttt{move(int, int)} by Richard, in the presence of the overloading method \texttt{move(Number, Number)}, originates a conflict, i.e. the semantic of both \texttt{move}s is different.

First, the source folders of each version are read, and an abstract syntax tree is generated for each class, collecting its members and bindings to one another. Based on this, the base and the original versions are compared, and the original and remote versions thereafter, i.e., the changes between these versions are calculated. The result is that method \texttt{public void reset() was added in the original version}, and method \texttt{public void move(int x, int y)} was added in the \texttt{remote} version (though it was detected in the merged version).

After that, the members in an object-oriented relationship are found, the result being: \texttt{public void move(Number x, Number y) overloads public void move(int x, int y)}. In here each of the members is checked against each other, applying rules to determine if they are in the said relationship. For these methods, the check if they overload each other returned \texttt{true}.

Then, the call graph is calculated. MC-MODS detects that the added method \texttt{public}
void reset() calls public void move(int x, int y), in the merged version.

With this information, it is deduced that the reset method is a target method, as it calls a member in an object-oriented relationship, and the test class in the shown box (4.5) is generated.

```java
import static infrastructure.main_locator.*;
import infrastructure.main.*;
import infrastructure.others.*;
import infrastructure.signatures.*;
import java.util.List;
import junit.framework.TestCase;
import org.junit.Test;

public class TestGenReset1 extends TestCase {
    @Test(expected = CompareException.class)
    public void test() throws Exception {
        JavaMethodSig target = new JavaMethodSig("com.wilhelm.pei", "Shape", 1,"void", "reset");
        RootClassLoader rl = new RootClassLoader(Object.class.getClassLoader(),
                C:/Users/Ricardo/runtime-PEI/OriginalOverloading/bin", "com");
        RootClassLoader rl2 = new RootClassLoader(Object.class.getClassLoader(),
                C:/Users/Ricardo/runtime-PEI/MergedOverloading/bin", "com");
        List<CompareException> l = Template.test(target, rl, rl2);
        printErrors(l);
        assertFalse(l.isEmpty());
    }
}
```

At this moment, the generated test is being run by JUnit. Next, I describe what happens from here on.

First, the target method is initialized, with the added method (by Garfield): public void reset() in class Shape. If this method had parameters, they would be initialized as well. The constructors we can see of the JavaMethodSig class receive the package name of the class, the class name, its modifier, and its return type and method name, both in String. The modifiers are represented in integers is to simplify conversions between the integers and the huge number of possibilities for combinations of the enum elements of the java.lang.reflect.Modifier class, such as public static final synchronized, represented by the integer 12345.

After that, the two classloaders of each side are created. They are initialized with the absolute path to the bin folder of each project, calculated in the class MainHandler, and the name of the containing package. The test method, Template.test(), is then called, receiving the target method and both classloaders as parameters.

Going into the functionality of this method, it starts by finding reset() and its class, via reflection, returning an instance of the class java.lang.reflect.Method and java.lang.Class, respectively, so the method can be invoked via reflection. The returned class is class com.wilhelm.pei.Shape, obtained with the first classloader, rl. Next, the constructor for this class is retrieved. Class com.wilhelm.pei.Shape has constructor Shape(int x, int y) and Shape(Point p). As the constructors are retrieved in alphabetical order, and by number of parameters in ascending order,
Chapter 4. Design and Implementation

Shape(int x, int y) is chosen (the fact that 'i' of int is before 'p' of Point is valued more than the number of parameters of the first constructor being greater). The parameter generation for this constructor is simple, as both parameters are of type int, therefore they are randomly generated with java.util.Random.nextInt(). The values 5 and 7 are obtained, as we will see in the next section.

Then, the first instance is created with the constructor and its generated parameters, wrapped with Transloader and cloned into the second instance using the second classloader, r2.

After this, the parameters are generated for the target method. As reset does not have any, it is just invoked via reflection, on the original side. On the merged side, the target method is invoked, using an instance of the class transloader.InvocationDescription mentioned earlier.

Now, invoking reset on the original side calls move(Number x, Number y), moving it to the origin, (0,0). On the merged side, move(int x, int y) is invoked, which adds the values (0,0) to the shape's coordinates, not changing its location, so it still has the coordinates (5,7).

Finally, the two objects of class Shape are compared. As (0,0) and (5,7) are different, a CompareException is returned and added to the list, and the method Template.test finishes.

This method returns a list of CompareExceptions, which are thrown when a conflict is found. The exceptions/errors within are then printed by the method printErrors, located in class Locator, and then the assertion method junit.framework.Assert.assertFalse() is invoked, to make sure that the list is empty, which means conflicts do not exist. If they do exist, the test fails. In this case, because the list is not empty, the CompareException is printed, and the conflict is reported, as we can see in the following section.

We can see that the conflict has manifested itself because the method move—called by reset and draw—is move(Number, Number) in the original version, and move(int, int) in the merged version, and because both moves have different semantics and behavior.

Here is an example of a reported conflict for the case of Overloading. The list returned by the method Template.test() returned this CompareException, meaning that the two objects given to compare are different so the invocation of the target method, public void com.wilhelm.pei.Shape.reset(), resulted in two different objects, which are then printed. The class Shape implements the toString() method, and it outputs the value of its two attributes, x and y, of type int. The first Shape represents the Shape

```java
((...\RemoteOverloading\src\TestGenReset1)
CompareException: Compare objects are different for method 'public void com.wilhelm.pei.
 Shape.reset()'
(Shape: (0, 0), Shape: (5, 7))
```
in the original version and the second represents the one in the merged version.
Chapter 5

Conclusion

This thesis develops a technique, partially implemented by a tool named MC-MODS, to handle the problem of detecting behavioral conflicts in Java, in the context of software development using version control systems.

The designed solution is based on two ground ideas: that merging operations of version control systems can be viewed as mutation operations performed by other team members on the source code of the person in charge of the merging process; and that these mutations, in an object-oriented language like Java, are the main causes for behavioral conflicts. The solution consists in: (a) the definition of a catalog of semantic conflicts that may arise from merging collaborative work, based in part on the fault models and existing object-oriented mutation operators [8, 13]; (b) the creation of test templates to represent the identified mutation types; and (c) the development of a test generation technique to detect these behavioral conflicts. An important feature of the solution is that the tool is correct—no false positives are reported. However, it does not detect all possible semantic conflicts so it is not complete.

In other words, there are three possible outcomes when analyzing a merge: (a) it detects the mutation and the conflict, (b) it detects the mutation but not the conflict, (c) it detects neither one.

The tool is implemented as a plug-in for the Eclipse IDE, since this platform is widely known for its extensibility, the support for plug-ins and other add-ons, and it is very often used with version control systems, further supporting software integration.

The functionality of the tool consists in two parts: the identification of methods which semantic could have been affected by the merging process; and the automatic generation and execution of tests to confirm the presence of behavioral conflicts, which are then reported.

I believe that the mutation operations identified in the catalog cover most object-oriented features that may cause behavioral conflicts in Java, and that they help to diagnose conflicts that emerge from software integration as early as possible. Furthermore, the implementation of MC-MODS facilitates the production of future test cases, as this
tool is easily extensible with new rules that the conflict detection mechanisms are based in.

Although the test generation of the tool is implemented in a minimalist way, other methods are possible, and it will always be necessary to balance the degree of coverage and the cost of its implementation.

As MC-MODS is a prototype, there are several of its functionality aspects that can be improved. The dimension of the changes present in each mutation scenario in the catalog is small, so testing with cases with bigger change sets would be important. This could be done by using larger Java applications and by combining different types of mutation operations at the same time. Every case in the catalog is projected for a team of two members, Garfield and Richard. Development teams are almost never so small, so MC-MODS would need to be tested with more team members in each case.

Another aspect is the point of view, that was focused on Garfield. In this work, Garfield is always the one making the changes, and Richard always introduces the mutation. In a more real case scenario, all developers may introduce changes and mutation operators, even both at the same time. This would increase the complexity of the conflict detection mechanisms involved, but there are ways to facilitate the resolution of the problem, such as continuous integration with messages, that inform developers where in the code their co-workers are currently making changes [3]. To include more methods with more parameters would also be interesting, in order to improve the quality of the parameter generation functionality.

The call graph implemented in this work is also capable of facilitating the conflict detection, as it contains information about which members call which members. Therefore it can be used, in the future, to detect binding changes, the main reasons for conflicts.

This plug-in is meant to be integrated into the update process of a version control system, detecting behavioral conflicts right after the user checks out the changes by other team members from the repository. This could be implemented with via a hook, a custom script that can be fired off when important actions occur—like the update and the merge processes. A client-side hook is the appropriate measure, as this specific type of scripts is used for client operations like committing and merging.

Another possible improvement to the functionality of MC-MODS would be to prevent the acceptance of source code with compilation errors by the plug-in, as such code certainly causes errors in the abstract syntax tree and/or in the change set, for instance, it would be useful to compile the needed source code and verify if it is free of errors before handing it to the plug-in. It would be easy to implement a hook to verify there are no errors in the code, or use a version control system, or configure one in such a way, so that it does not accept errors.

The detection mechanisms were not implemented successfully for one case, the inverse of Attribute Hiding (see Figure 3.8), due to the limitations of the Transloader tool.
It does not support cloning an object into another, when the class of the latter has more attributes than the first, as the tool probably does not generate default values for the missing attributes. One solution could be to create objects from custom classes that do not have the missing attributes.

Some conflicts captured in the catalog that were tested later were identified as non-conflicts and have been removed from the catalog. They include the inverses of Overloading [3.2] and Access Change 1 [3.10].

It is assumed that the operations in the catalog, that represent changes in behavior, do not contain equivalent mutants. This type of mutants cannot be caught with test cases, because they reveal behaviors that do not differ, so they . The inverses and reverses of each case are also not equivalent. Even if they were, each conflict in the catalog contains different mixes of object-oriented features, which strengthens the assumption that equivalent mutants are not present.

In conclusion, MC-MODS is an ongoing process, nevertheless an important step in minimizing behavioral conflicts in Java, and I hope further studies could perfect this tool in order to make conflict detection a little bit easier to all developers.
Bibliography


