FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automatic Detection of Semantic Conflicts in Merge Commits via LLMs

Duarte Guedes Sardão

WORKING VERSION

Mestrado em Engenharia Informática e Computação

Supervisor: José Campos

February 4, 2024

Automatic Detection of Semantic Conflicts in Merge Commits via LLMs

Duarte Guedes Sardão

Mestrado em Engenharia Informática e Computação

February 4, 2024

Resumo

Em desenvolvimento de software colaborativo, trabalho paralelo, desenvolvido em diferentes ramos, tem de ser frequentemente integrado. Devido às diferenças do trabalho efetuado nos diferentes ramos, conflitos surgem frequentemente. Alguns conflictos são simples de detetar e rectificar, como conflitos de integração textuais, que surgem quando diferentes desenvolvedores editeam a mesma linha em ramos diferentes: a maior parte de sistemas de controlo de versão consegue automaticamente detetar que a mesma linha foi alterada e impele o integrador a decidir numa solução: que alteração manter e que discartar.

Entre conflitos de integração, os semânticos emergem com um tipo particularmente difícil de resolver, porque não são detetados por sistemas de controlo de versão e, não tendo erros sintáticos, compilam com sucesso. Um exemplo de um conflito semântico pode ser visto numa classe Point, que contêm um método para calcular a distância para outro Point: num cenário em que um desenvolvedor num ramo A muda o cálculo de distância (de euclideana para manhattan, por exemplo), enquanto que outro chama o método dentro de uma função de movimento num ramo B, percebemos que depois de a integração nenhum erro é levantado, mas o comportamento está desviado dos ramos originais: especificamente, o movimento comportar-se-á de maneira diferente do que esperado no ramo B.

Procuramos explorar como o emergente ramo de Modelos de Linguagem Grande podem fornecer um enorme avanço na nossa abilidade de testar o comportamento alterado, introduzido ou perdido devido a conflitos semânticos. Especificamente, com base em trabalho anteriore que identifica prováveis conflitos e gera outputs, numa linguagem de domínio específica, analisamos a abilidade de ChatGPT para gerar testes unitários apropriados that evidenciam a presença de conflitos semânticos, com prompting adequado.

Abstract

In collaborative software development, parallel work done by several different branches often has to be merged. Due to the differences of work done in the difference branches, often conflicts arise. Some conflicts are easy to detect and rectify, such as textual merge conflicts, where different developers have altered the same line in different branches: most version control systems can detect that the same line has been changed and urge the merger to decide on a solution, for example, by keeping one change and discarding the other.

Among merge conflicts, semantic merges arise as particularly difficult to resolve, as they avoid detection by version control systems and lacking syntactic errors, compile successfully. An example of a semantic conflict can be seen in a class such as Point, containing a method to calculate the distance to another Point: in a scenario where one developer in branch A changes the distance calculation (from euclidean to manhattan, for example), while another calls the distance method for a movement function in branch B, we find that upon a merge while no errors are raised, the code exhibits altered behaviour from the original branches: specifically, the movement will be different from what was developed in branch B.

We seek to explore how the emerging field of Large Language Models can provide a breakthrough in our ability to test for the altered, introduced and lost behaviours arising from semantic conflicts. Specifically, building upon previous work that identifies probable conflicts and generates outputs, on a domain-specific language, we analyse the ability of ChatGPT to generate appropriate unit tests that highlight the presence of semantic conflicts, given appropriate prompting.

Acknowledgements

Aliquam id dui. Nulla facilisi. Nullam ligula nunc, viverra a, iaculis at, faucibus quis, sapien. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Curabitur magna ligula, ornare luctus, aliquam non, aliquet at, tortor. Donec iaculis nulla sed eros. Sed felis. Nam lobortis libero. Pellentesque odio. Suspendisse potenti. Morbi imperdiet rhoncus magna. Morbi vestibulum interdum turpis. Pellentesque varius. Morbi nulla urna, euismod in, molestie ac, placerat in, orci.

Ut convallis. Suspendisse luctus pharetra sem. Sed sit amet mi in diam luctus suscipit. Nulla facilisi. Integer commodo, turpis et semper auctor, nisl ligula vestibulum erat, sed tempor lacus nibh at turpis. Quisque vestibulum pulvinar justo. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nam sed tellus vel tortor hendrerit pulvinar. Phasellus eleifend, augue at mattis tincidunt, lorem lorem sodales arcu, id volutpat risus est id neque. Phasellus egestas ante. Nam porttitor justo sit amet urna. Suspendisse ligula nunc, mollis ac, elementum non, venenatis ut, mauris. Mauris augue risus, tempus scelerisque, rutrum quis, hendrerit at, nunc. Nulla posuere porta orci. Nulla dui.

Fusce gravida placerat sem. Aenean ipsum diam, pharetra vitae, ornare et, semper sit amet, nibh. Nam id tellus. Etiam ultrices. Praesent gravida. Aliquam nec sapien. Morbi sagittis vulputate dolor. Donec sapien lorem, laoreet egestas, pellentesque euismod, porta at, sapien. Integer vitae lacus id dui convallis blandit. Mauris non sem. Integer in velit eget lorem scelerisque vehicula. Etiam tincidunt turpis ac nunc. Pellentesque a justo. Mauris faucibus quam id eros. Cras pharetra. Fusce rutrum vulputate lorem. Cras pretium magna in nisl. Integer ornare dui non pede.

O Nome do Autor

"You should be glad that bridge fell down.
I was planning to build thirteen more to that same design"

Isambard Kingdom Brunel

Contents

List of Figures

List of Tables

Abbreviations and Symbols

LLM	Large Language Model
VCS	Version Control System
DSL	Domain Specific Language

Chapter 1

Introduction

The purpose of this chapter is to introduce the motivation for the work, briefly describe the problem at hand and outline the work that will be developed, as well as the structure of the thesis.

1.1 Motivation

Collective software development requires the handling of merge conflicts, as conflicts between parallel work arise. Called merge conflicts, as they arise when this parallel work is merged, they vary in their difficulty of detection. Common textual conflicts, where the same line is altered by multiple people, are automatically detected by version control systems, allowing amendments to be easily made. However not all conflicts are this easily detected and their introduction bring with it the addition of software bugs to the system. Semantic merge conflicts, in particular, are hard to detect, both by software and human review and remain a hard to solve situation. Given that around 30% of developers do not even actively monitor for merge conflicts and of those who do, mostly do it with reactive strategies [15], the ability to automatically detect these would be a great boon for the field of software development, especially as by developers own admissions, the longer a merge conflict is left unsolved, the harder it becomes to resolve it: “Untangling takes days instead of minutes when it gets too out of hand.” [15].

The recent revolution in the field of Large Language Models (LLMs) may prove to add a valuable tool in tackling this.

1.2 Problem

To manage the concurrent work of several developers in software projects, it is common to employ *version control systems* (hereby referred as *VCS*), which can be defined as “a system that manages the development of an evolving object. In other words, it is a system that records any changes made by the software developers.” [30].

A significant task of version control is managing access to shared resources. In the “classic scenario” a lock-modify-unlock paradigm was adopted, where a given file would be locked for modification while it is being modified, thus ensuring each resource can only be handled by one actor at a time. VCS's however, generally implement a copy-modify-merge mechanism: concurrent work can done on a resource, with joining the parallel work together handled by merges afterwards, with two "branches" of work merged into one [16].

Merge conflicts arise when parallel work cannot be clearly merged. Of this, several different types exist, as summarized by Tom Mens [14]:

Textual conflicts occur when the same textual elements of code are modified in both branches of a merge. For example, when the same line of code is modified by two people in their respective branches.

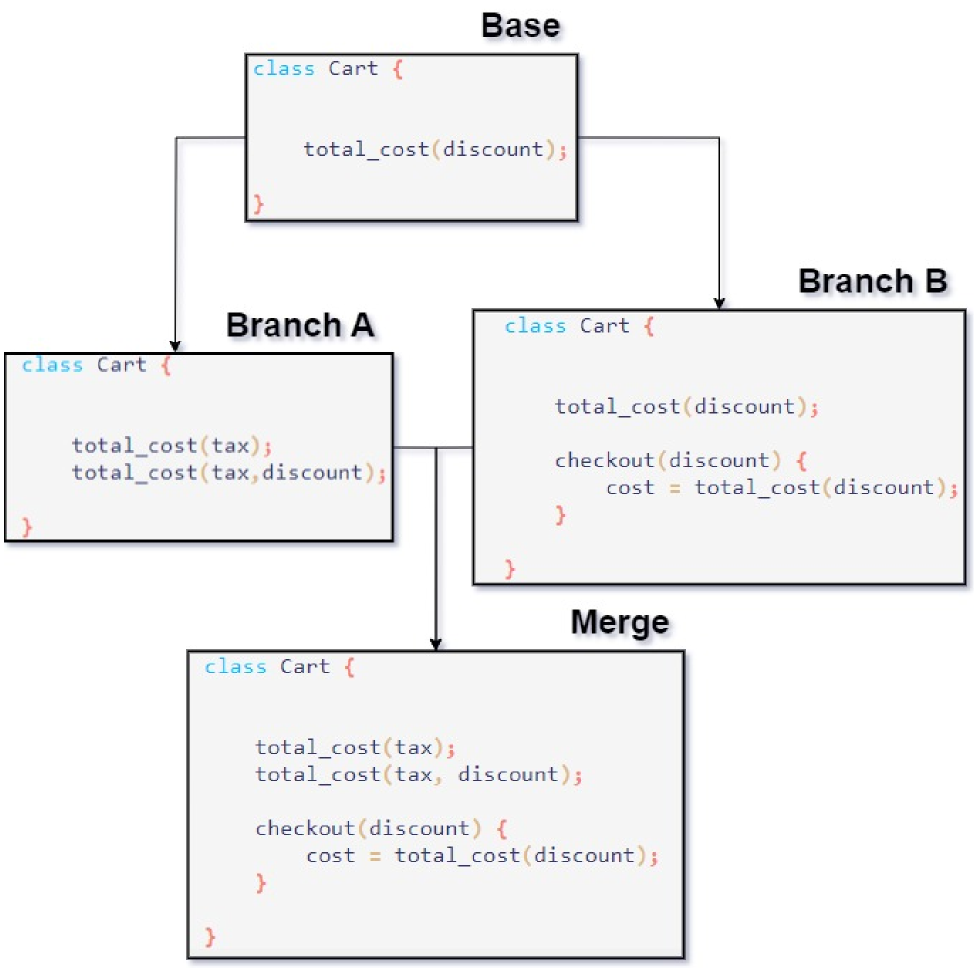
Syntactic conflicts arise from parallel changes that when merged do not generate textual conflicts, but the resulting merge creates code that is invalid given the languages rules. For example, programmer A renames a variable, while programmer B uses the variable somewhere (with the original name). There is no textual code, but the code will not compile due to the usage of an uninitialized variable.

Finally semantic conflicts occur when parallel changes do not trigger any conflict and are syntactically valid, but the resulting code doesn not behave as expected, or exhibits lost or new unexpected behaviour.

Most VCS's, such as Git, implement textual merge tools (ergo, they can only identify textual conflicts), however there are specialized tools that handle other types of merges: for example Turbomixer for syntactic merges [14]. This focus on textual merging is generally fine as around 90% of conflicts are textual [18] and syntactic conflicts are easy to identify after merges, as errors will be clearly indicated and programs will not compile. Semantic conflicts remain as both undetected by VCS's and hard to detect after merges. Thus, identifying methods to automatically identify and highlight semantic conflicts in merge commits has been a persistent problem and a source of study in this field [3] [22] [23].

A motivating example of a merge conflict can be seen in a class "Cart", for a shopping app. Initially having just a method total_cost that calculates the total cost given a percentage of discount, it is modified in branch A: a additional parameter for tax and the method is overloaded with another implementation that just calculates tax and not discount. Then, in branch B, a checkout method is added, which calls the original total_cost method. After merging, no error appears, but the checkout function will be running the total_cost tax method, expecting the calculation of a discount. The example is shown in figure 1.1.

From here the conflict arises, as after merging the behaviour is altered: the checkout function which would calculate the cost with a discount, now instead uses the value given for discount to calculate tax. To identify this conflict, we would need to generate a test that would find evidence of this alteration: in this case finding that the checkout function is charging more money than expected.



```
class Cart {

    total_cost(discount);
}
```

```
class Cart {

    total_cost(tax);
    total_cost(tax,discount);
}
```

```
class Cart {

    total_cost(discount);

    checkout(discount) {
        cost = total_cost(discount);
    }
}
```

```
class Cart {

    total_cost(tax);
    total_cost(tax, discount);

    checkout(discount) {
        cost = total_cost(discount);
    }
}
```

Figure 1.1: Model of the motivating example conflict, with Base, Branch A, Branch B and Merge versions of the code.

1.3 Goal

The goal of the presented project is to develop an automated system that can detect possible merge conflicts in a merge and generate the necessary unit tests to identify it using a LLM, allowing for easier detection of this type of conflicts to be integrated into the version control workflow of developers.

1.4 Approach

In previously developed work, a Domain-Specific Language was designed which describes the changes made in a three-way merge [3]. This DSL provides a summarized report of a three-way merge. We will feed the DSL reports into ChatGPT and induce it to generate unit tests based on them which should help assess the presence of a semantic conflict originating from the merge.

1.5 Thesis Structure

The first chapter introduces the topic and problem at hand. The second chapter details existing related work on the topics at hand. The third chapter introduces the preliminary work. The fourth chapter gives a conclusion.

Chapter 2

Related Work

The following chapters introduces the two main topics dealt with in this dissertation (merge conflicts and the usage of large language models for software verification and test generation). They explore related work and how we can build on it to develop our approach.

2.1 Semantic Conflict Detection

The study of merge techniques and conflicts has a long history, likely even predating the specific terminology itself. Thus there is a large corpus of work to explore

Several related works exist which seeks to develop methods that can more systemically identify adulterated behaviour arising from semantic conflicts.

2.1.1 Detection without Testing

Several solutions exists which attempt to identify the presence of semantic conflicts without generating unit tests. An example of this rests on verifying overriding assignment: if a merge was successful where the same variable was assigned in both A and B, it's likely this is unintentional and there is a conflict. Thus, the solution checks if such a situation happens and reports if so to the developers [1].

Static analysis solutions have been considered, implementing the previous overriding assignment checks, inter procedural data flow and program dependence graph, which try to find data and control flows between the changes done by both developers, as well as confluence, which checks if there is data and control flows from both branches that flow into some common path. This static analysis shows much better F1 score and recall than dynamic analysis techniques, but much worse precision [4].

The tool DeltaImpactFinder compares the impact of a change in the origin and destination branches, calculating the difference, the "Delta-Impact". To measure impact, dependencies are mapped out: the assumption is that when an entity is changed it's dependencies are impacted. A semantic conflict is identified if there are missing or extra dependencies and this can be identified with the calculation of the delta-impact [5].

The notion of semantic conflict-freedom is seen as a sufficient condition for correctness. The SafeMerge [24] tool seeks to verify the presence of semantic conflict-freedom in the merge process, avoiding the creation of semantic conflicts. Semantic conflict-freedom can be defined, for branches O (Origin), A, B and M (Merge), for all i inputs:

- If out(O,i) $\neq$ out(A,i), then out(M,i) = out(A,i)
- If out(O,i) $\neq$ out(B,i), then out(M,i) = out(B,i)
- Otherwise out(O,i) = out(A,i) = out(B,i) = out(M,i)

2.1.2 Generation of Unit Tests

In identifying the presence of semantic conflicts in a merge conflict, developed solutions have focused on the automatic generation of unit tests.

By Da Silva et al, we find an attempt at identifying cases of semantic conflict by applying automated behaviour change detection [22]. In summary, with a base commit B, a left L, right R and merge M, they observe that a generated unit test that passes in L but fails in B partially reveals the effect of the changes made in that branch. If the test then fails in M, it is likely the changes made in R interfere. To generate unit tests they used EvoSuite and Randoop [17] [8].

In their analysis they found that the developed tool only detected interference in four out of 15 changes within merge scenarios that do actually suffer from interference, corresponding then to a recall of 0.267. While this is a very modest rate, it displayed no false positives (precision of 1) and thus could likely be integrated in a testing process to prune possible merge conflicts early, or further studied and refined.

Building upon their previous work, Da Silva et al [23] proposed SAM (SemAntic Merge), a tool that generates tests upon merges in Java. It can be found at `https://github.com/leusonmario/SAM`. In summary, SAM initially does a simple textual merge to integrate the difference branches while identifying possible textual conflicts. After merging four program versions are built, to fully describe the merge scenario under test: Base, Left, Right, Merge. Source code transformations to improve testability are also part of the process, but optional. Finally, the test generation tools are fed objects serialized during the execution of existing test suites. After applying four test generations tools: EvoSuite [8], Differential EvoSuite, Randoop [17] and Randoop Clean, their own adapted version of Randoop, SAM executes the generated tests against the four versions of the program, identifies which tests failed, interpreting it with pre-defined interference criteria heuristics and from there reports conflicts, if detected [23].

Detecting 9 out of 28 conflicts, it shows improvements over previous work: the authors specifically highlight the best performance when combining tests from only EvoSuite and Differential EvoSuite. Regarding behaviour changes specifically, 89 are found. In both cases, they highlight the ability of transformations (for example, making private fields public) to increase testability, showing moderate improvements in some tested scenarios, with 20 additional changes being detected and 3 additional conflicts detected (Differential Evosuite detects 3, each other generator detect one conflict each). Regarding the 19 false-negatives, 11 of them showed behaviour changes, which were not caused by the changes due to the semantic conflicts. [23].

Nuno Castanho has proposed the tool UNSETTLE (aUtomatic uNit teSt gEneraTion for semanTic confLict dEtection) [3]. This tool, which can be found at `https://github.com/conflito/unsettle`, is composed of two modules:

Changes-Matcher identifies the possible presence of semantic conflicts, by first computing the changes between different versions (base and variants) and then comparing it to a set of patterns (listed on table 2.1 describing common sources of conflicts as a base. From this it generates a DSL file, highlighting which methods and classes should be put under test to identify the conflict.

The second module is the test generator, a modified version of EvoSuite that takes the previously created artifact as an input to guide test generation.

Of particular interest to us is Changes-Matcher, as this is also the starting point for our work, with the usage of a LLM over EvoSuite for test generation instead.

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

Table 2.1: Semantic Conflict Patterns identified by Changes-Matcher [3].

Ti Jao et al have proposed test oracles for program merges. Most significantly, not only do they support two and three way merges but also octopus merges. The developed tool, TOM (testing on merges), generates tests to identify unexpected and lost behaviour. For this, they implement diff-line as a criteria, guiding the tests to cover lines that have been modified between the different program versions, generating different assertions for different versions. Stability checking is done: the test is rerun five times before being handed to the developers. The tool identifies 45 three-way and 87 octopus merges, from a universe of 389 of each [11].

Table 2.2 shows a comparison between some key features of the presented solutions.

Tool	Recall	Octopus Merge Support	Publicly Available	Test Generator
SAM	43% (12 out of 28)	No	Yes	EvoSuite, Differential EvoSuite, Randoop, Randoop Clean
UNSETTLE	35% (6 out of 17)	No	Yes	EvoSuite
TOM	17% (132 out of 778)	Yes	No	EvoSuite

Table 2.2: Comparison of SAM, UNSETTLE and TOM tools [23][3][11]

2.2 Test Generation

2.2.1 Traditional Automatic Test Generation

Traditional methods of automated test generation can be divided into random and search-based techniques. The former, being random, is simpler and faster, while the latter employs heuristics and search algorithms to fulfill some criteria, like maximizing code coverage.

Randoop is one such example of random test generation. Some of the disadvantages of this technique can be mitigated with the employment of feedback-direction [17]. Effectively, the search space of possible random tests is pruned, by guiding the test generator towards valid cases, avoiding expansion on invalid test cases [17].

EvoSuite implements search-based automatic test generation for Java code, aiming for tests that achieve high coverage, while being as small as possible and providing assertions. To achieve this they implement both evolutionary search to evolve the suite with respect to a coverage criterion and mutation testing to generate assertions [8].

EvoSuiteR extends EvoSuite, aiming to provide automated generation of regression tests. Thus it takes account two versions of the software and aside from coverage, it considers state distance (“how different is the state of all objects in the test suite across the two versions”) and control flow distance (“how far are the two versions from diverging”) [20].

The symbolic execution technique, while introduced in the 70s has been more explored as computing power increased. It executes programs with symbolic rather than concrete values: assignments are represented as functions and conditionals as constraints. From this, we can identify specific paths and branches. This can be done statically, we symbolic execution techniques first used to derive all paths or dynamically, with symbolic execution being updated throughout execution, allowing us to find alternatives to the current branch [9].

Similar suites have been developed, for example, for Python [13].

2.2.2 Test Generation with LLM's

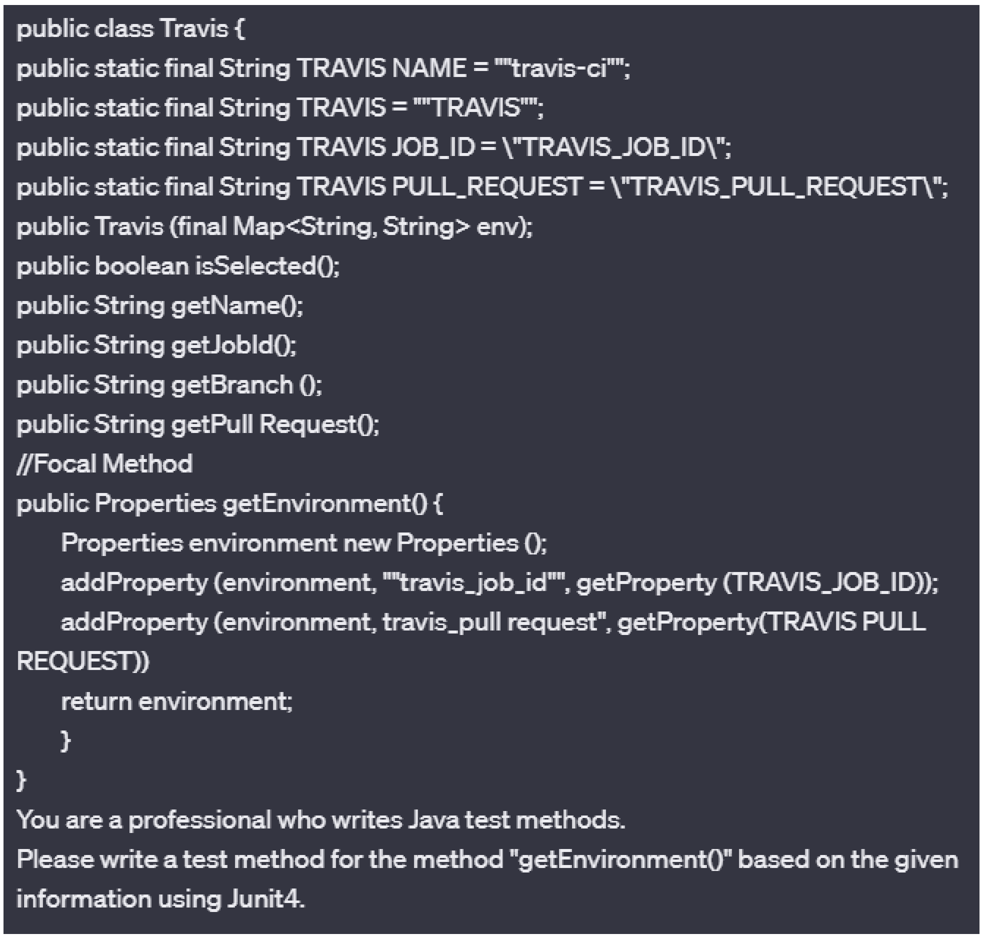
The recent explosion in complexity and popularity of LLM's has suscitated developer interest in their abilities with regards to accelerate and automate software engineering. Angela Fan et al identified that by 2023 3% of pre-prints were related to Large Language Models and 11% of those related to their use in software engineering[6]. Particularly relevant is their ability to generate tests, with an expectation that they could achieve better coverage, correctness and readability than previous techniques of automated test generation[26]. In comparison to traditional suites for automated test generation, such as EvoSuite, Palus, Randoop, and JTExpert, ChatGPT has shown, given right tuning of temperature settings, to evidentiate equivalent robustness [10]. From surveys done on the topic, we find Codex and GPT variants to be the most commonly used LLM's for this issue [26].

While many studies suggest LLM's are competitive with traditional methods of software generation, we can also find evidence to the contrary. Specifically, testing across several LLM's, including Codex, StarCoder and GPT-3.5-Turbo, shows they fail in all regards compared to EvoSuite, primarily due to the generation of non-compilable code, often due to "hallucination" of non-existent types and methods [21]. Yutian Tang and others find EvoSuite outperforms ChatGPT in code coverage [25]. Given the novelty of the technology, it is unsurprising there is such variance in reported results and research, but it is worth investigating.

Despite variable conclusions regarding correctness and coverage, it seems generally agreed that ChatGPT is good at generating readable code. Yutian Tang et al highlight that despite minor style errors, primarily in identation, the generated code is clear and easy to understand[25]. A survey of software developers has found ChatGPT to have comparable and even better readability than manually written tests [28].

Given the LLM can produce unreliable results, the maximization of ChatGPT's abilities with regards to test generation has been explored: techniques such as prompting the LLM for a explanation of what the code is intending to do [12] and feeding error messages from codes that fail to compile or execute as intended back to the LLM for correction [28] have shown an amazing capacity for test generation, given the right prompting.

Regarding prompting, understanding how to best engineer prompts to achieve the desired output from the LLM is a particularly relevant issue. For example, in the generation of test inputs, research found "generate diverse test input" to be preferable over "generate test inputs that result in different outputs between PUT and reference versions", as ChatGPT could not accurately identify the nuances required to correctly carry out the latter's instructions. [12] With unit testing generation in particular, Zhiqiang Yuan et al, in developing the ChatTester tool, highlight the importance of combining a natural language descriptor with the appropriate code context [28]. The prompt using, showing this can be seen on figure 2.1. Another technique utilised is asking the LLM to infer the intention before developing the tests, as seen in figure 2.2 which leads it to generate correct tests where basic prompting would fail [28].

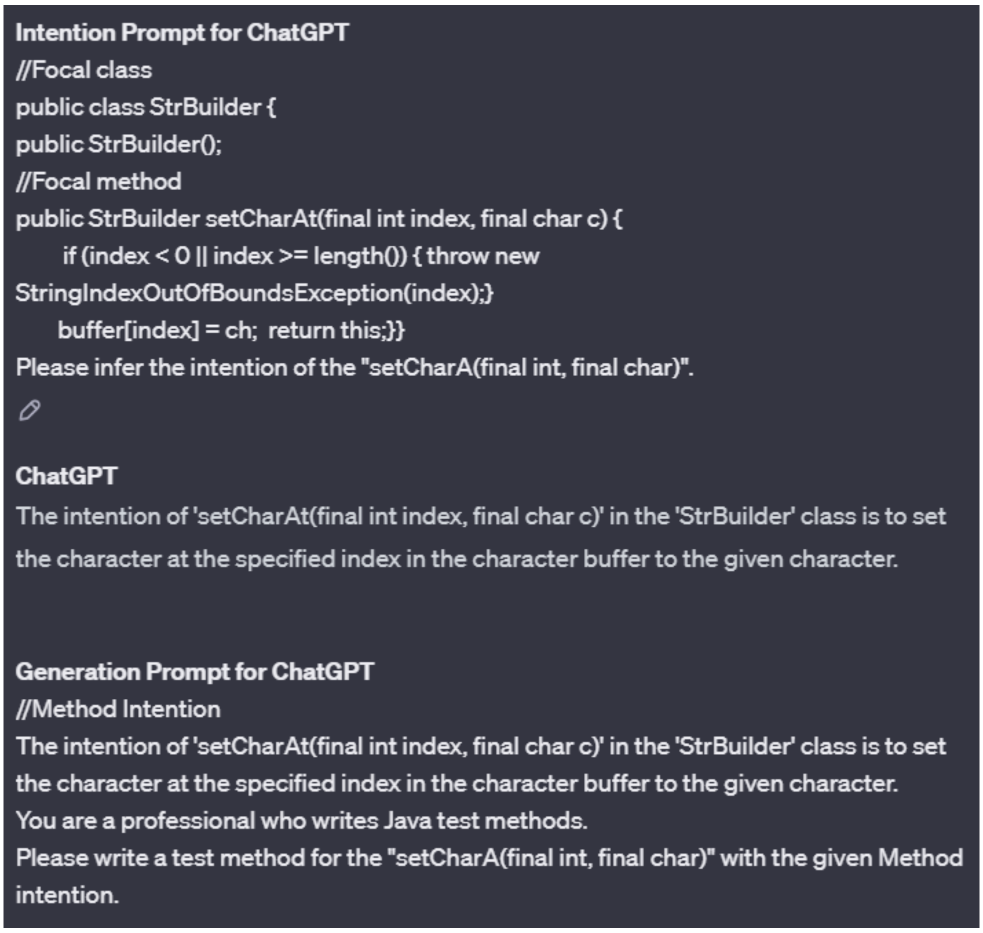


```
public class Travis {
public static final String TRAVIS NAME = ""travis-ci"";
public static final String TRAVIS = ""TRAVIS"";
public static final String TRAVIS JOB_ID = \"TRAVIS_JOB_ID\";
public static final String TRAVIS PULL_REQUEST = \"TRAVIS_PULL_REQUEST\";
public Travis (final Map<String, String> env);
public boolean isSelected();
public String getName();
public String getJobId();
public String getBranch ();
public String getPull Request();
//Focal Method
public Properties getEnvironment() {
    Properties environment new Properties ();
    addProperty (environment, ""travis_job_id"", getProperty (TRAVIS_JOB_ID));
    addProperty (environment, travis_pull request", getProperty(TRAVIS PULL
REQUEST))
    return environment;
    }
}
You are a professional who writes Java test methods.
Please write a test method for the method "getEnvironment()" based on the given
information using Junit4.
```

Figure 2.1: Basic Prompt in ChatTester. Code context and natural language instructions can be identified. [28].

While they argue that it is little known how effective these strategies are, J.D. Zamfirescu-Pereira and others highlight several prompting strategies: give examples of desired interaction, write prompts that look somewhat like code and repeat yourself [29]. They note difficulties participants have in prompt generation, such as avoiding giving examples due to fear the LLM will simply replicate it, difficulty in searching online for help and adapting existing solutions and over-generalizing from a single example [29].

In prompting we can distinguish a few types of prompts: zero-shot, where instructions are simply given, few-shot, where examples of inputs and outputs are given and few-shot with preamble, combining the previous two examples to give both a preamble instructing the LLM on what


Intention Prompt for ChatGPT
//Focal class
public class StrBuilder {
public StrBuilder();
//Focal method
public StrBuilder setCharAt(final int index, final char c) {
    if (index < 0 || index >= length()) { throw new
StringIndexOutOfBoundsException(index);}
    buffer[index] = ch; return this;}}
Please infer the intention of the "setCharA(final int, final char)".

ChatGPT
The intention of 'setCharAt(final int index, final char c)' in the 'StrBuilder' class is to set the character at the specified index in the character buffer to the given character.

Generation Prompt for ChatGPT
//Method Intention
The intention of 'setCharAt(final int index, final char c)' in the 'StrBuilder' class is to set the character at the specified index in the character buffer to the given character.
You are a professional who writes Java test methods.
Please write a test method for the "setCharA(final int, final char)" with the given Method intention.

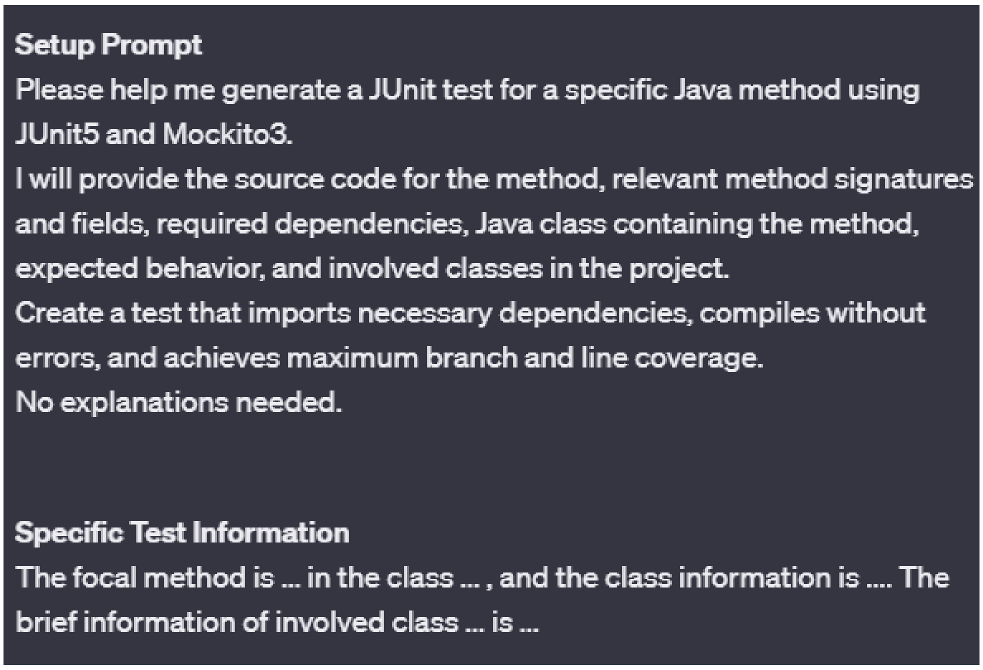
Figure 2.2: Inference Prompt in ChatTester [28].

to achieve, follow by examples of inputs and outputs [7]

Given the natural language aspect of LLM prompting is one the features that most distinguishes it from classical programming, it is worth exploring methods of systematizing it, making it work more like code. It is with this objective that LMQL (Language Model Query Language) was developed, introducing a scripting based query language [2]. Evidence shows it reduces computing costs by up to 80% [2].

The team behind ChatUniTest presents a two step prompting system, where instructions of what is to be done and how are first provided [27], which can be seen in figure 2.3.

The training of LLM's in open source repositories gives cause to the fear that automatic test generation might be reliant on the model being trained in the code under test, or simply replicating existing testing suites. To assuage this, it was identified that even when generating tests for packages hosted in GitLab (which were not used to train the LLM under study, gpt3.5-turbo), they still maintained high coverage. In addition, it was found that 60.0% of the tests had <= 40% similarity to existing tests and 92.8% had <= 50%. Thus we can conclude the LLM is not simply copying tests from the training set without alteration[19].



Setup Prompt
Please help me generate a JUnit test for a specific Java method using JUnit5 and Mockito3.
I will provide the source code for the method, relevant method signatures and fields, required dependencies, Java class containing the method, expected behavior, and involved classes in the project.
Create a test that imports necessary dependencies, compiles without errors, and achieves maximum branch and line coverage.
No explanations needed.

Specific Test Information
The focal method is ... in the class ... , and the class information is .... The brief information of involved class ... is ...

Figure 2.3: ChatUniTest two-step prompting [27]

2.3 Conclusions

The related work aids our own work in several ways: the work explored as regards unit test generation can provide show us a baseline functioning of the idea we are trying to implement. Particularly, we can use their statistics regarding percentage of conflicts found to assess whether our own solution is an improvement with regards to previous work.

Crucially important too is the work of Nuno Castanho and, as previously mentioned, the developed Changes-Matcher [3].

The study of works on LLM test generation should also prove useful in the future, allowing us to get a better understanding of their functioning and capabilites, as well as providing ideas and prompt generation techniques which we may have to apply in the future.

Chapter 3

Preliminary and Future Work

3.1 Preliminary Work

In the initial step of work, we superficially explored ChatGPT's ability to generate tests for an example conflict of Point, where a distance method is altered from euclidean to manhattan in one branch and in the other branch, a move method is changed from using the value 1 for x and y movement to using the result of the distance calculation.

We tested two frameworks, first just asking for a test, with prompts based on the testing indications given by the DSL for the case:

- A Dependency Based semantic conflict was possibly introduced in a 3-way merge. Develop a test for the class Point, that covers the methods move() and distance(), without calling distance() directly. Before the merge, the class under test was: [base Point] After the merge, it was: [merged Point].
- A Dependency Based semantic conflict was possibly introduced in a 3-way merge. Develop a test for the class Point, that covers the methods move() and distance(), without calling distance() directly. Before the merge, the class under test was: [base Point] In the branch A it was changed to: [A Point] In the branch B it was changed to: [B Point] After the merge, it was: [merged Point].

For this, the LLM simply took one version and created tests taking it as correct behaviour. In the first case, for Base and in the second for Merge. This is not ideal, as the first does not allow us to distinguish if the behaviour changed due to merging, or just do to changes in the branches. The latter takes merge as correct behaviour and will thus always fail.

Other tests involved first asking for an explanation if there was a merge conflict there, before asking for a test

- We have done a merge on a piece of code. Before the merge, the code was: [code]

  In the branch A it was changed to: [code]

  In the branch B it was changed to: [code]

  After the merge it was: [code]

  Do you believe there could be a merge conflict here? Where? Explain why.

- We have done a merge on a piece of code.

  Before the merge, the code was: [code]

  In the branch A it was changed to: [code]

  In the branch B it was changed to: [code]

  After the merge it was: [code]

  Do you believe there could be a merge conflict here? Where? Explain why. Pay attention to both additions and modifications.

- We have done a merge on a piece of code code.

  Before the merge, the code was: [code]

  In the branch A it was changed to: [code]

  In the branch B it was changed to: [code]

  After the merge it was: [code]

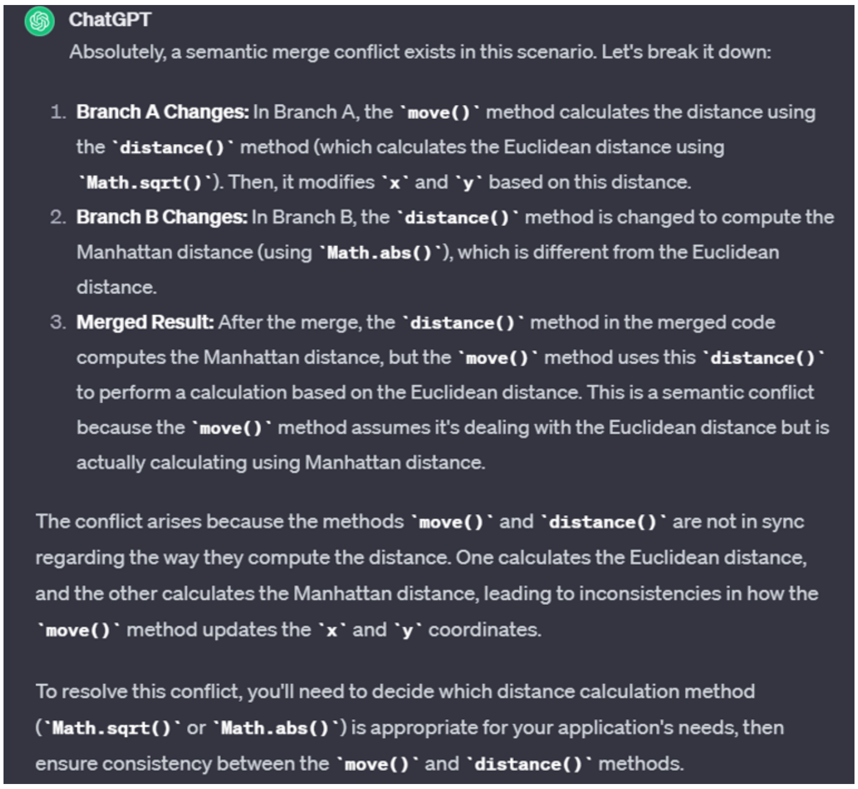
  Do you believe there could be a *semantic* merge conflict here? Where? Explain why. Pay attention to both additions and modifications.

The first two prompts failed. Indeed for prompt ChatGPT only mentioned the changes in move, but even after prompting it to pay attention to distance, it failed to identify the conflict. For both, it seemed to not understand what was being referred to when speaking of conflict, describing all changes as a "conflict". Thus it would say, there is a conflict between whether to use euclidian and manhattan distance and there is a conflict on how the move function works.

The last prompt given, highlighting the *semantic* aspect of the merge conflict, yielded good results, with the conflict being perfectly described, as seen in figure 3.1. After prompting for a test, it generates an appropriate one: it tests if the move function updates Point appropriately based on the euclidean distance. Thus it fails for base and branch B, where movement is still done with the value of 1, it passes for branch A as there move is based on distance; and it fails in merge, as now movement is being done with the distance calculation, but the calculation is now manhattan.

In simple fabricated scenarios, where simple conflicts were added to existing software solutions, ChatGPT showed ability to identify and describe the semantic conflict. Despite this, test generation remained complicated and few of the successful identification of semantic conflicts yielded working tests. The final prompt follows.

> Generate a Junit unit test to identify this semantic conflict, knowing what you do now.
> The test must compile without errors and require no further alterations. It should require no further dependencies and import all classes correctly.



ChatGPT

Absolutely, a semantic merge conflict exists in this scenario. Let's break it down:

1. **Branch A Changes:** In Branch A, the `move()` method calculates the distance using the `distance()` method (which calculates the Euclidean distance using `Math.sqrt()`). Then, it modifies `x` and `y` based on this distance.
2. **Branch B Changes:** In Branch B, the `distance()` method is changed to compute the Manhattan distance (using `Math.abs()`), which is different from the Euclidean distance.
3. **Merged Result:** After the merge, the `distance()` method in the merged code computes the Manhattan distance, but the `move()` method uses this `distance()` to perform a calculation based on the Euclidean distance. This is a semantic conflict because the `move()` method assumes it's dealing with the Euclidean distance but is actually calculating using Manhattan distance.

The conflict arises because the methods `move()` and `distance()` are not in sync regarding the way they compute the distance. One calculates the Euclidean distance, and the other calculates the Manhattan distance, leading to inconsistencies in how the `move()` method updates the `x` and `y` coordinates.

To resolve this conflict, you'll need to decide which distance calculation method (`Math.sqrt()` or `Math.abs()`) is appropriate for your application's needs, then ensure consistency between the `move()` and `distance()` methods.

Figure 3.1: ChatGPT description of the semantic conflict

While adapted to avoid common pitfalls, the tests generate still suffered from basic issues such as missing imports, which can be mitigated by prompting the LLM for correction automatically. More complex issues of implementation, such as calling the base function instead of dependent, wrong usage of construction and function returns, parameter types or unnecessary mocks. Further, when prompting for correction, the LLM often explains that the developer should correct this, thus further work needs to be done to ensure the tool does all the work itself.

Prompting in real-world examples, or more complex fabricated examples, has produced far worse results. Several refinements were made to the prompts, most significantly: the usage of "git diff" to highlight the specific changes in branch A and branch B, the explanation of the conflict present and the specification of the target method where the conflict is evident. These modifications were largely unsuccessful and did not lead to identification of any conflicts.

We also had to reckon with size limits for messages. Thus, we started with an explanatory prompt and then feed the information step by step. However, it remained crucial to remind the LLM of the goal in the last message.

> We have done a merge on a piece of software and introduced a semantic conflict of types: "Update two different dependencies of a method or update one method and concurrently update one of its dependencies" and "Concurrent changes to the same method". I will now show the base commit, the diff in branch A, the diff in branch B,

> and the final merge version in 4 separate messages. At the end I want you to explain why and where the semantic conflict is present.

Common issues are confusion between textual and semantic merges, which can be mitigated by clear explanation of what a semantic merge is; hallucinations of features not present in the code or hallucinations of changes where changes were not made; lack of focus on the methods where conflict is evident, despite reiterations; loss of focus when prompt has to be split into several messages. When the tools do understand the semantic conflict, the best description they can give is to point out the changes made between branches followed by the suggestion that there may be a conflict there. Part of the difference between real-world examples and fabricated scenarios may come down to the information given. The fabricated scenarios, based on work of Nuno Castanho [3], were accompanied with a description of the specific semantic conflict present. Thus it is possible collecting and offering that information with real-world scenarios may improve the ability of LLMs in this regard.

Another factor in consideration is the issue of dependencies, as so far testing had focused on just a unitary class. Given that semantic conflicts can involve interactions between classes and subclasses or other dependencies, it is relevant to provide further information. Initial tests just added one dependency, wether by calling a class methods or due to a inheritance relationship. An example, with the addition of an illustrative example of a semantic conflict follows.

> We have done a merge on a piece of software and introduced a semantic conflict of type Parallel Changes in Method.
>
> Semantic conflicts occur when concurrent and syntactic-correct changes in different regions of a source file or different files cause the software system to misbehave. For example, suppose there is a Java class ‘Point‘ with a method ‘distance()‘ that computes the Euclidean distance of a Point to the origin and Bob decides to modify ‘distance()‘ so it computes instead the Manhattan distance. At the same time, Alice, not aware of Bob’s changes, creates a new method ‘move()‘ that uses ‘distance()‘ to calculate the Euclidean distance. Then, the changes of both developers are merged. As Bob and Alice did not modify the same lines of code, there is no textual conflict. There is neither a syntactic conflict as the merged code still compiles. However, the program now has an unexpected behaviour. The ‘move()‘ method introduced by Alice no longer moves a Point an Euclidean distance (as Alice was expecting) but rather moves a Manhattan distance
>
> The affected declaration is copyWithDefaults().
>
> A first message will detail the class before the merge, the diffs for both branches and the class after the merge. A second message will have a dependent class, whose methods indirectly call copyWithDefaults(). After I send these next two classes, identify and describe the semantic conflict.

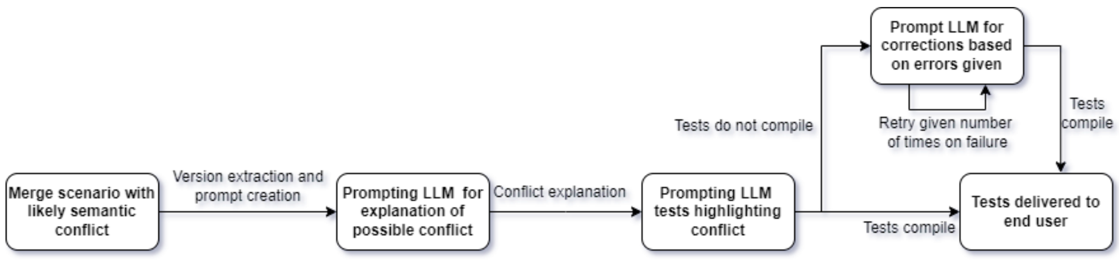

Merge scenario with likely semantic conflict
Version extraction and prompt creation
Prompting LLM for explanation of possible conflict
Conflict explanation
Prompting LLM tests highlighting conflict
Tests do not compile
Prompt LLM for corrections based on errors given
Retry given number of times on failure
Tests compile
Tests compile
Tests delivered to end user

Figure 3.2: Functioning pipeline of proposed tool

A more complex evolution on this idea consisted of providing textual representations of UML graphs, such as plantUML structure graphs. These however proved to be complex in their own regard as they often induced the LLM to "forget" previous information and its' goal, possibly due to their large size.

No successful tests have been generated for real world scenarios. Experiments with other LLMs, such as Bing AI or Bard, yield similar problems and new ones, such as more stringent size limits, suggesting an altered approach may have to be taken.

3.2 Future Work

3.2.1 Development Plan and Tool Functioning

Development of the solution will go through several stages. Firstly, the process which has already been in progress, is the initial evaluation of LLM's and prompt techniques. After acquiring the list of subjects to test and deciding on the LLM, we can systematize this to develop the prototype solution. From here we start developing a tool that can automatically generate a prompt, get a test from the LLM and then run it. This prototype tool can then be further augmented, by applying corrections for tests that may fail to compile. Figure 3.2 details the expected functioning of the tool once completed. A final period of evaluation and observations will compare our results to previous work and reflect on possible further improvements.

3.2.2 Subjects

To assess the validity of a developed solution, a collection of subjects to test must be collected. While several previous work has compiled collections of merge commits with semantic conflicts, the collection done by Nuno Castanho [3] is particularly useful, being publicly available, closely related to our own work, and also allowing us to draw direct comparisons. Most importantly, it aggregates merge instances from both Silva et al [22] and Sousa et al [24], thus being more extensive.

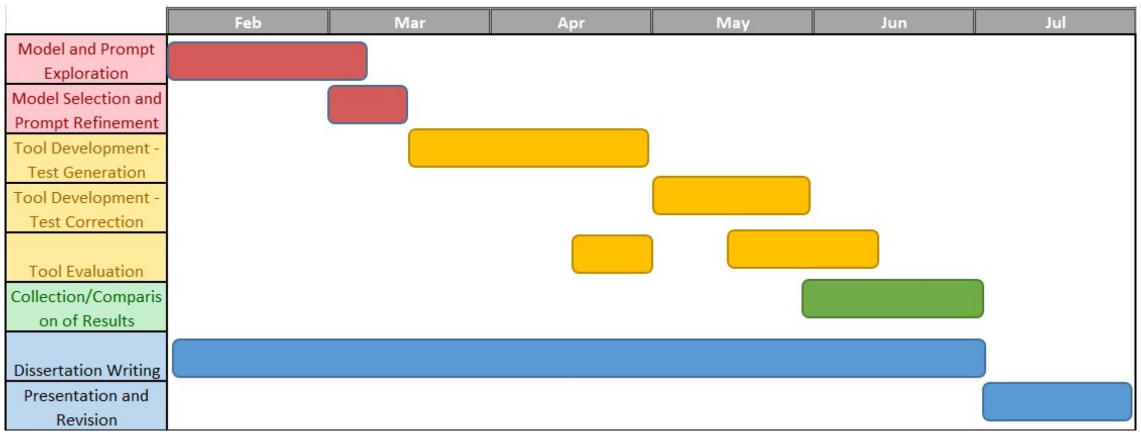


Figure 3.3: Gantt diagram of working plan

3.2.3 LLM's

While preliminary work sought to explore ChatGPT and Llama (both CodeLlama and Llama 2), hardware constraints meant we were unable to explore Llama. ChatGPT, being hosted online for free, did not suffer from such issues, but future problems may arise, most notably the possibility of the size limit for messages interfering in our work. Going forward we will further explore how to mitigate these and possible alternatives.

Chapter 4

Conclusion

In our preparation for dissertation we explored and presented the problem of software merges, more specifically, the difficulty in establishing an automated method to detect, identify and highlight semantic conflicts that arise.

We explored related work that can provide avenues of study and structures to guide the development of our solution. Specifically, previous work on test generation to identify semantic conflicts, the state of the art of automated test generation and the usage of Large Language Models for test generation, with specific focus on the important of appropriate prompting as well as output correction.

Forward, we will apply this knowledge in developing a solution that can hopefully aid developer workflows by identifying and generating tests for semantic conflicts.

References

[1] Matheus Barbosa, Paulo Borba, Rodrigo Bonifacio, and Galileu Santos. Semantic conflict detection with overriding assignment analysis. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, SBES '22, page 435–445, New York, NY, USA, 2022. Association for Computing Machinery.

[2] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[3] Nuno Castanho. Semantic conflicts in version control systems, 2021. Available at https://repositorio.ul.pt/handle/10451/50658.

[4] Galileu Santos de Jesus, Paulo Borba, Rodrigo Bonifácio, and Matheus Barbosa de Oliveira. Detecting semantic conflicts using static analysis, 2023.

[5] Martín Dias, Guillermo Polito, Damien Cassou, and Stéphane Ducasse. Deltaimpactfinder: Assessing semantic merge conflicts with dependency analysis. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '15. ACM, July 2015.

[6] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large language models for software engineering: Survey and open problems, 2023. Available at https://arxiv.org/abs/2310.03533.

[7] Alexander J. Fiannaca, Chinmay Kulkarni, Carrie J Cai, and Michael Terry. Programming without a programming language: Challenges and opportunities for designing developer tools for prompt programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI EA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[8] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[9] Patrice Godefroid. Test generation using symbolic execution. In *Foundations of Software Technology and Theoretical Computer Science*, 2012.

[10] Vitor Guilherme and Auri Vincenzi. An initial investigation of chatgpt unit test generation capability, 2023. Available at https://dl.acm.org/doi/10.1145/3624032.3624035.

[11] Tao Ji, Liqian Chen, Xiaoguang Mao, Xin Yi, and Jiahong Jiang. Automated regression unit test generation for program merges. *Science China Information Sciences*, 65(9):199103, Aug 2022.

[12] Tsz-On Li, Wenxi Zongc, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting, 2023. Available at `https://arxiv.org/pdf/2304.11686.pdf`.

[13] S. Lukasczyk, F. Kroiß, and G. Fraser. An empirical study of automated unit test generation for python. *Empirical Software Engineering*, 28(36), 2023.

[14] T. Mens. A state-of-the-art survey on software merging, 2002. Available at `https://ieeexplore.ieee.org/document/1000449`.

[15] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. The life-cycle of merge conflicts: Processes, barriers, and strategies. *Empirical Software Engineering*, 24(5):2863–2906, 2019.

[16] Stefan Otte. Version control systems, 2009. Available at `https://www.mi.fu-berlin.de/inf/groups/ag-tech/teaching/2008-09_WS/S_19565_Proseminar_Technische_Informatik/otte09version.pdf`.

[17] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84, 2007.

[18] D.E. Perry, H.P. Siy, and L.G. Votta. Parallel changes in large scale software development: an observational case study. In *Proceedings of the 20th International Conference on Software Engineering*, pages 251–260, 1998.

[19] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation, 2023.

[20] Sina Shamshiri. Automated unit test generation for evolving software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 1038–1041, New York, NY, USA, 2015. Association for Computing Machinery.

[21] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. An empirical study of using large language models for unit test generation, 2023.

[22] Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and João Moisakis. Detecting semantic conflicts via automated behavior change detection, 2020. Available at `https://ieeexplore.ieee.org/document/9240661`.

[23] Léuson Da Silva, Paulo Borba, Toni Maciel, Wardah Mahmood, Thorsten Berger, João Moisakis, and Vinícius Leite Aldiberg Gomes. Detecting semantic conflicts with unit tests, 2023. Available at `https://arxiv.org/abs/2310.02395`.

[24] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. Verified three-way program merge. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[25] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation, 2023.

[26] Junjie Wang, Chunyang Chen Yuchao Huang, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language model: Survey, landscape, and vision, 2023. Available at `https://arxiv.org/abs/2307.07221`.

[27] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. Chatunitest: a chatgpt-based automated unit test generation tool, 2023.

[28] Zhiqiang Yuan, Mingwei Liu Yiling Lou, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation, 2023. Available at `https://arxiv.org/abs/2305.04207`.

[29] J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. Why johnny can't prompt: How non-ai experts try (and fail) to design llm prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. Version control system: A review. *Procedia Computer Science*, 135:408–415, 2018. The 3rd International Conference on Computer Science and Computational Intelligence (ICCSCI 2018) : Empowering Smart Technology in Digital Era for a Better Life.