Galileu Santos de Jesus

A Lightweight Technique for Detecting Semantic Conflicts with Static Analysis

Universidade Federal de Pernambuco
Pós-Graduação em Ciência da Computação
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2026

Galileu Santos de Jesus

A Lightweight Technique for Detecting Semantic Conflicts with Static Analysis

A Ph.D. Thesis presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science.

Area of Concentration: Software Engineering and Programming Languages

Advisor: Paulo Henrique Monteiro Borba

Co-advisor: Rodrigo Bonifácio de Almeida

Recife

2026

FICHA

Dedico esta tese a todos que amam ciência, mas acima de tudo, amam viver e enxergam a vida além do lattes. Sejam felizes no percurso!

ACKNOWLEDGEMENTS

À minha família, em especial a minha mãe Lenice, meu pai Gilvan, minhas irmãs e sobrinhos, que souberam sempre me acompanhar mesmo de longe, dividindo momentos muito importantes, aos meus amigos que estiveram ao meu lado durante este percurso. Ao meu amigo Kleber, por ter feito parte desta jornada me motivando sempre a continuar. Aos meus amigos Aldo, Adsson, Caique e Weldon, que sempre estiveram próximos. À minha futura esposa, que isto seja um exemplo de motivação para nossa família. Aos meus professores desde o ensino básico até o superior, aos que me mostraram o que é ciência e me orientaram na graduação e mestrado. Ao professor Paulo Borba pela inquestionável competência com a qual orienta seus alunos e pelas oportunidades oferecidas. Aos membros do SPG pelas experiências e conhecimentos compartilhados, bem como os bons e divertidos momentos vividos. Ao meu coorientador Rodrigo Bonifácio pela fundamental colaboração neste trabalho, você não tem noção do quanto foi importante nesta jornada! Aos membros de minha banca pela paciência, disponibilidade e comentários que certamente contribuirão para este e trabalhos futuros. Finalmente, agradeço à FACEPE por financiar minha pesquisa e ao Centro de Informática da Universidade Federal de Pernambuco por todo recurso que me ofereceu.

RESUMO

Palavras-chaves: substituir atribuição; conflitos de integração de código; desenvolvimento colaborativo.

ABSTRACT

Keywords: overriding assigment; code integration conflicts; collaborative development.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

As software projects evolve and become increasingly complex and collaborative, effectively managing code integration issues becomes a crucial task. Modern software development relies heavily on version control systems and parallel development workflows, where multiple developers simultaneously work on different features or bug fixes. While textual conflicts, inconsistencies that prevent automatic merging at the line level, are well-known and widely addressed by existing tools, they represent only the tip of the iceberg in terms of integration challenges.

In collaborative environments where multiple developers work on different parts of a codebase, more subtle inconsistencies can emerge that go undetected by traditional line-based merge tools. These issues, known as *semantic conflicts*, occur when changes that are syntactically correct and successfully merge nonetheless introduce unintended behaviors or logical inconsistencies in the integrated code (SARMA; REDMILES; HOEK, 2011; BRUN et al., 2013; TOWQIR et al., 2022; ZHANG et al., 2022; SUNG et al., 2020). Unlike textual conflicts, which are immediately visible during the merge process, semantic conflicts are insidious: they allow the merge to complete successfully, often pass compilation, but introduce subtle bugs that may only manifest during testing or, worse, in production.

Semantic conflicts can be classified based on when they are detected in the software development lifecycle. We adopt the following terminology, where conflicts are categorized according to the development phase in which they surface:

- **Merge conflicts:** Textual conflicts detected by version control systems during the merge process, based on line-level changes.
- **Build conflicts:** Syntactic and static semantic conflicts that cause compilation failures after merging.
- **Test conflicts:** Dynamic semantic conflicts detected by the test suite after merging.
- **Production conflicts:** Dynamic semantic conflicts that escape testing and manifest only in production environments.

Textual merge tools, by design, operate on a line-by-line basis and cannot recognize logically incompatible changes that are not contiguous in the source code. For instance, consider a scenario where one developer modifies a method's signature while another developer, working in a different file or distant location in the same file, adds a call to the original version of that method. The code merges cleanly without any textual conflicts, yet the integrated version fails to compile due to the method signature mismatch. This represents a *build conflict* (SARMA; REDMILES; HOEK, 2011; BRUN et al., 2013; TOWQIR et al., 2022; ZHANG et al., 2022; SUNG et al., 2020), which can be detected through static analysis or compilation.

A more challenging category involves *dynamic semantic conflicts*, which are not detected by either textual merge tools or static analysis tools like compilers. These arise when developers make changes that interact in subtle ways, modifying program behavior without causing compilation errors. For example, if one developer changes the initialization value of a shared variable while another adds code that depends on the original value, the merged code compiles successfully but exhibits incorrect runtime behavior (HORWITZ; PRINS; REPS, 1989; YANG; HORWITZ; REPS, 1992; SHAO; KHURSHID; PERRY, 2009; BRUN et al., 2013; PASTORE; MARIANI; MICUCCI, 2017; FILHO, 2017; SOUSA; DILLIG; LAHIRI, 2018; SILVA et al., 2020; ZHANG et al., 2022).

Throughout this thesis, we focus primarily on *dynamic semantic conflicts*, which we refer to simply as *semantic conflicts*. These conflicts arise when developers modify or use the same *state elements*, program entities that hold or represent the program's state during execution. Such elements include any variable or data structure that can be read or written at runtime. In Java, this encompasses *instance fields*, *static fields*, *local variables*, *method parameters*, and *array elements*. A semantic conflict occurs when a state element is modified by one developer and subsequently used (directly or indirectly) by another, resulting in an *interference* between their changes.

1.1 THE PROBLEM AND RESEARCH CHALLENGES

Dynamic semantic conflicts can negatively impact development productivity and the quality of software products, particularly in projects with divergent forks (SUNG et al., 2020; ZHANG et al., 2022), but also in projects with a single remote repository. These conflicts are particularly problematic because they can escape detection during code review and testing, leading to bugs that manifest in production environments where they are more costly to fix and may affect end users.

Traditional approaches to detecting semantic conflicts face several fundamental challenges:

Testing-based approaches can identify some semantic conflicts by running the project's test suite on the merged code and checking for newly failing tests (SILVA et al., 2020; SILVA et al., 2023; SILVA et al., 2024). However, these approaches suffer from several limitations. First, they have low recall – many semantic conflicts do not cause test failures because test suites rarely achieve complete coverage. Second, not all test failures in merged code indicate semantic conflicts. For instance, consider a scenario where variables `l` and `r` are initialized as `0` in the base version. Developer Left changes the code to initialize `l` with 1, while developer Right changes it to initialize `r` with 1 and adds a test that asserts `r==1 && l==0`. This test passes in Right's version and fails in the merged version, but there is no semantic conflict – both developers changed unrelated variables independently.

Formal verification approaches using theorem proving (SOUSA; DILLIG; LAHIRI, 2018) can provide strong guarantees about the absence of conflicts, but they are computationally expensive and require significant expertise to apply. Additionally, they often need formal specifications of intended behavior, which are rarely available in practice.

Traditional static analysis approaches based on System Dependence Graphs (SDGs) (BINKLEY; HORWITZ; REPS, 1995; FILHO, 2017) can detect interference between developers' changes by analyzing data and control dependencies across all three versions involved in a merge (base, left, and right). While these approaches can be effective, they are computationally expensive because they require constructing and analyzing SDGs for multiple versions of the entire codebase, limiting their scalability to large projects.

Given these limitations, there is a clear need for practical, scalable techniques that can detect semantic conflicts with reasonable accuracy while being efficient enough for integration into standard development workflows.

1.2 RESEARCH APPROACH AND CONTRIBUTIONS

This thesis investigates the use of *lightweight static analysis* techniques for detecting semantic conflicts in software merges. The key insight behind our approach is that we can approximate interference detection by analyzing only the merged version of the code, which is automatically annotated with information indicating which instructions were modified or added by each developer. This eliminates the need to analyze multiple versions and construct complex intermediate representations like SDGs for each version, significantly reducing computational costs.

Our research addresses three fundamental questions about lightweight static analysis for semantic conflict detection:

1. **Feasibility:** Can lightweight static analysis techniques effectively detect semantic conflicts by analyzing only the merged version with developer attribution information?
2. **Optimization:** How do different lightweight static analysis techniques and their combinations compare in terms of accuracy (precision and recall) and computational efficiency?
3. **Comparison:** How do lightweight static analysis techniques compare to traditional approaches based on SDGs in terms of both effectiveness and efficiency?

To answer these questions, this thesis presents three main contributions, each corresponding to a research study:

First contribution (Chapter 3): We present a novel lightweight static analysis technique for detecting semantic conflicts that operates solely on the merged version of the code with developer attribution. We propose four analysis variants – Interprocedural Direct Flow, Intraprocedural Direct Flow, Control Dependence, and Program Dependence Graph – that identify data and control flow dependencies between instructions changed by different developers. We conduct an empirical evaluation comparing our lightweight approach to existing techniques based on testing, formal verification, and SDG-based static analysis. The results demonstrate that lightweight static analysis can detect semantic conflicts efficiently, though with trade-offs between different metrics.

Second contribution (Chapter 4): Building on the initial lightweight approach, we conduct a comprehensive investigation of how different lightweight static analysis techniques can be combined to optimize performance. To support this investigation, we implement nine distinct static analyses and systematically explore both their individual effects and their possible combinations. We evaluate the accuracy and computational efficiency of these configurations, revealing that specific combinations can significantly improve precision while maintaining reasonable recall. Our study also identifies optimal configurations for different development scenarios, depending on whether teams prioritize precision, recall, or a balanced trade-off.

Third contribution (Chapter 5): We present an empirical comparison between lightweight static analysis techniques and traditional SDG-based approaches. While SDG-based techniques have been proposed as theoretically more comprehensive, their practical advantages over lightweight techniques remain unclear, particularly when considering the trade-off between accuracy and computational cost. Our evaluation provides insights into when each approach is most appropriate and how they complement each other in practice.

Together, these contributions provide a comprehensive understanding of lightweight static analysis for semantic conflict detection, from initial feasibility through optimization to comparison with existing state-of-the-art techniques. The insights from this research enable development teams to integrate semantic conflict detection into their workflows with appropriate configurations based on their specific needs and constraints.

1.3 THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

Chapter 2 provides the necessary background on version control, merge conflicts, and existing approaches to semantic conflict detection. It establishes the terminology and concepts used throughout the thesis and surveys related work in this area.

Chapter 3 presents our first contribution: a lightweight static analysis technique for detecting semantic conflicts. This chapter describes the approach in detail, presents the empirical evaluation methodology, and discusses the results demonstrating the feasibility of lightweight analysis.

Chapter 4 presents our second contribution: a systematic comparison of different lightweight static analysis techniques and their combinations. This chapter explores the design space of lightweight analyses and identifies optimal configurations for different scenarios.

Chapter 5 presents our third contribution: an empirical comparison between lightweight static analysis and SDG-based approaches. This chapter provides insights into the trade-offs between these different analysis paradigms.

Chapter 6 discusses related work in semantic conflict detection, program analysis, and software merging, positioning our contributions within the broader research landscape.

Chapter 7 concludes the thesis by summarizing our main findings, discussing limitations, and outlining directions for future work in semantic conflict detection and analysis.

2 BACKGROUND AND MOTIVATION

3 A LIGHTWEIGHT TECHNIQUE FOR DETECTING SEMANTIC CONFLICTS WITH STATIC ANALYSIS

4 COMPARING STATIC ANALYSES FOR IMPROVED SEMANTIC CONFLICT DETECTION

5 EMPIRICAL EVALUATION OF SDG AND LIGHTWEIGHT STATIC ANALYSIS FOR DETECTING SEMANTIC CONFLICTS IN SOFTWARE MERGES

6 RELATED WORKS

7 CONCLUSIONS

7.1 FUTURE WORK

REFERENCES

BINKLEY, D.; HORWITZ, S.; REPS, T. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, ACM New York, p. 3–35, 1995.

BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, IEEE, p. 1358–1375, 2013.

FILHO, R. S. M. d. B. *Using information flow to estimate interference between same-method contributions*. Dissertação (Mestrado) — Federal University of Pernambuco, 2017.

HORWITZ, S.; PRINS, J.; REPS, T. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, ACM, p. 345–387, 1989.

PASTORE, F.; MARIANI, L.; MICUCCI, D. Bdci: Behavioral driven conflict identification. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. [S.l.: s.n.], 2017. p. 570–581.

SARMA, A.; REDMILES, D. F.; HOEK, A. V. D. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, IEEE, p. 889–908, 2011.

SHAO, D.; KHURSHID, S.; PERRY, D. E. Sca: a semantic conflict analyzer for parallel changes. In: *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. [S.l.: s.n.], 2009. p. 291–292.

SILVA, L. D.; BORBA, P.; MACIEL, T.; MAHMOOD, W.; BERGER, T.; MOISAKIS, J.; GOMES, A.; LEITE, V. *Detecting Semantic Conflicts with Unit Tests*. 2023.

SILVA, L. D.; BORBA, P.; MACIEL, T.; MAHMOOD, W.; BERGER, T.; MOISAKIS, J. Detecting semantic conflicts with unit tests. *Journal of Systems and Software*, 2024.

SILVA, L. D.; BORBA, P.; MAHMOOD, W.; BERGER, T.; MOISAKIS, J. Detecting semantic conflicts via automated behavior change detection. In: *2020 IEEE International Conference on Software Maintenance and Evolution*. [S.l.: s.n.], 2020. p. 174–184.

SOUSA, M.; DILLIG, I.; LAHIRI, S. K. Verified three-way program merge. *Proceedings of the ACM on Programming Languages*, ACM, 2018.

SUNG, C.; LAHIRI, S. K.; KAUFMAN, M.; CHOUDHURY, P.; WANG, C. Towards understanding and fixing upstream merge induced conflicts in divergent forks: An industrial case study. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. [S.l.: s.n.], 2020. p. 172–181.

TOWQIR, S. S.; SHEN, B.; GULZAR, M. A.; MENG, N. Detecting build conflicts in software merge for java programs via static analysis. In: *37th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2022. p. 1–13.

YANG, W.; HORWITZ, S.; REPS, T. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology*, ACM, p. 310–354, 1992.

ZHANG, J.; MYTKOWICZ, T.; KAUFMAN, M.; PISKAC, R.; LAHIRI, S. K. Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper). In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. [S.l.: s.n.], 2022. p. 77–88.

APPENDIX A

MANUAL ANALYSIS PROTOCOL FOR DEFINING THE GROUND TRUTH

To define the ground truth, a double-check procedure must be performed:

- For each scenario, a pair of reviewers is assigned to conduct the manual analysis following the instructions below.
- The manual analysis must be carried out individually by each collaborator assigned to the scenario.
- The individual analyses of the two collaborators must then be compared:
    - If they converge, the result must be adopted as the scenario's ground truth.
    - If they diverge, the collaborators must meet and discuss the scenario together in order to reach a consensus.
        - If consensus is not possible, the scenario must be submitted to a third collaborator, who will make the final decision.

INSTRUCTIONS FOR PERFORMING THE MANUAL ANALYSIS TO DEFINE THE GROUND TRUTH

The dataset consists of a number of subjects or cases to be analyzed.
Each subject is a pair composed of a merge scenario (a tuple with base, left, right, and merge commits) and a method or field declaration (yes, we also consider fields, not only methods) that was independently modified by two branches (or two developers).

A scenario that contains only a single declaration modified by two branches results in a single subject (and, consequently, a single row in the dataset spreadsheet).
A scenario *s* that contains *n* declarations modified by two branches results in *n* subjects:

(s, declaration 1), …, (s, declaration n)

Double check that you have the latest version of the dataset (https://github.com/spgroup/mergedataset) in your machine.

The dataset consists of a number of subjects, or cases to be analyzed. Each subject is a pair formed by a merge scenario (tuple with base, left, right, and merge commits) and a method or field (yes, we do not focus only on methods) declaration that was independently changed by two branches (or two developers). A scenario that contains just a single declaration that was changed by two branches leads to a single subject (and, consequently, a single row in the dataset spreadsheet). A scenario *s* that contains *n* declarations that were changed by two branches leads to *n* subjects: (*s*,declaration 1), …, (*s*,declaration *n*).

For each subject (*s,d*) in the dataset, the manual analysis should follow these steps:

1. Use kdiff to understand the changes made to base in the left and right branches
    a. Just load the base, left and right commits of *s* into kdiff (http://kdiff3.sourceforge.net) (kdiff3 left.java base.java right.java)
    b. Look for the declaration *d* that was changed in both branches
    c. Focus the analysis on this declaration
    d. We consider the left branch to be the sequence (path) of commits from the left commit up to base; similarly for the right branch
2. Take one or more screenshots that reveal the changes in the declaration that was the focus of the previous step, and how they are integrated into the merge commit
    a. Running the commands: kdiff3 left.java base.java right.java
    b. Add the screenshots to this presentation (http://gg.gg/imagesmerge)
    c. Considering the order in which the merge scenario appears in the dataset spreadsheet, add one slide with the merge commit hash
    d. Right after that, add one or more slides with the screenshots of the diff between left-base-right (this order is important)
    e. Following that, add a screenshot of the merge commit with "`// left`" and "`// right`" annotations at the end of the lines changed by the left and right branches, respectively
        i. If a line is changed by both left and right, add "`// left and right`"
        ii. The focus of the screenshots should be on revealing the changes in the declaration, but if the declaration is short enough the screenshots should show the whole declaration
    f. If the changes in left and right lead to a merge conflict
        i. Please add a note ("Merge conflict") to the slide
        ii. If the files in the merge commit still have the conflict markers (that is, the conflicts were not resolved), try to create a new commit that removes the markers and incorporates the changes from both left and right in a logical way
            - If this is not possible, the subject should be discarded.

        - Otherwise, keep going with the merged files in the new merge commit you created to fix the conflicts, and add its screenshot to the presentation as in item "d" (the presentation will have screenshots for the original and the fixed merge)

    g. If the changes in the left and right branches are not incorporated in the merge commit (some of the changes might have been removed or adapted by the integrator for a number of reasons: resolve a merge conflict, resolve a detected build, test or production conflict, remove redundancy, etc.), or are incorporated but the merge commit has additional changes made by the integrator (that is, some changes do not come from the left and right branches)
        i. Please add a note ("Merge amended by the integrator") to the slide
        ii. Replay the merge (git checkout left-hash; git merge right-hash), and check that the result is different from what is stored in the original repository
        iii. The manual analysis should proceed considering both merge commits (the original one in the repository, and the replayed one), with annotated screenshots for both (use "`// integrator`" if a line was changed by the integrador; if a line was changed by left, right and the integrator, add the three names to the comment)
            - The analysis of the original one will help us understand whether there is locally observable interference caused by the changes made by left, right and the integrator (so possibly an interference that was not caused by the changes in left and right, but by the changes made by the integrator)
            - The analysis of the replayed one will help us to understand whether there is locally observable interference caused by the changes made by left and right (so certainly an original interference caused by left and right changes)

3. Summarize the changes made in the left branch as observed in kdiff
    a. The focus is on what kdiff shows, not on all changes made by commits in the left branch
    b. So if an early commit in the branch adds, for example, an assignment to variable x (say `x=1`), and a later commit removes that, we won't see that in Step 1 above; the summary should also not mention this change, as the focus is on the end result
    c. The summary should be expressed in terms of
        i. The performed actions: added, removed, changed
        ii. The syntactic changes made to program elements: class, field, constructor, method declarations, initialization blocks, statements, expressions, arguments and parameters, if and while conditions, then branch, else branch, method body, loop body, annotations, comments
        iii. As in the following styles: "Left removes the reference to the field *maxRetries,* in the string expression that is returned by method *toString*";

"Left changes the condition of an if statement in method *outerHtmlHead.* The then branch of the if calls method *preserveWhitespace,* which is now only called if the parameter node is not null and is an instance of Element"

d. To decide what should go in the summary, think about a toy example that would replicate the essence of the differences in the original subject. The summary should then describe such essence, using the vocabulary in the item above.

4. Summarize the changes made in the right branch as observed in kdiff (similar to the previous item)
5. Now, analyzing the merge commit, check whether there might be a Direct Flow (DF) from the execution of the changes in left to the execution of the changes in right, or vice-versa
   a. We consider that there might be such a flow when the changes (additions and modifications, we do not consider the effect of removals on DF) in one of the branches might semantically (that is, its execution) involve a **writing operation** (assignment, file write operation, method call with side effect, etc.) to a **state element** (local variable, parameter, static or instance field, file, stream, expression in return statement, exception raised, etc. So including temporary state too) that is transitively associated with a **reading operation** (usage reference in an expression or argument, file reading operation, etc.) involved in the changes (additions and modifications) made by the other branch
   b. As DF analysis involves understanding the execution of code changed by left or right, we also have to analyze the behavior of methods and constructors called by the declaration under analysis. So, for example, the code "`x=0;\\left ... C.m(x);`" where the whole `C` is marked as right has a DF only if the body of `m` actually reads the value stored in `x`; so we have to analyze the body of m in this case. If the call to `m` is also marked as right (as in "`x=0;\\left ... C.m(x); \\ right`"), then we have a DF no matter how the body of `m` refers to its parameter, since the execution of the method call itself involves computing the values of all arguments (as Java is not a lazy language).
   c. We are interested in detecting the possibility of a DF, not a guarantee of DF. So in "`x=0;\\left ... if (cond) C.m(x); \\ right`" we consider that there might be a DF, even though we are not sure whether `cond` can be eventually true. If we are sure that `cond` is always false, we don't say that there might be DF in the considered subject.
   d. The analysis is **local**, in the sense that we consider only the code of the declaration under analysis and of the methods and constructors that are called in its body. We do not consider the **global** surrounding context formed by the parts of the system that call the method or field declared by the declaration under analysis.
   e. Similarly, the integrated changes might affect different state elements (say window.size and window.color) that lead to a single combined visual effect on the system GUI, which is a (intended or unintended) globally observable interference. However, as our analysis is local, we focus only on the effects on the state

elements described before, which does not include the GUI visual state that is rendered by some GUI framework that accesses the non visual state elements (say window.size and window.color) that were changed. As the analysis is local, we don't even know if the changed state elements will be considered by such a framework.

f. For the reasons above, we consider only the effects related to the semantics of Java. So we consider that changes involving annotations have no semantic impact, as analysing such impact would typically require a global analysis for understanding the behavior of tools and frameworks that explore the annotations.

g. As DF must involve at least a **writing operation** and a **reading operation**, if the changes in both branches involve only read operations, or involve only write operations, or are only whitespace changes, you can quickly know that there is no DF

h. To establish if there is a DF, do not think about the current implementation of DF, simply follow the conceptual definition of DF as in the previous items

i. Think of the system state as the contents of its RAM, file system, input and output streams, what we can see in its display, etc. Think of a state element as a part of that, like a specific value stored in one of the RAM addresses. Assuming that, we consider "`a[6]`" refers to address 123, for example, of the RAM. Executing "`a[6] = 0`" changes this state element, changing the value stored in the RAM address 123. When we then execute "`b=a;b[6]=5`" we again affect what is stored in RAM address 123, that is, we affect the same state element as before. When we execute "`b=revert(a);a=b;a[6]=4`" we are now affecting another state element, another RAM address, assuming that revert returns a new array.

j. To analyze cases that have a line or statement marked as left and right (which could be caused by both branches making the exact same change in a line, or the integrator resolving a merge conflict by incorporating changes from both branches in the same line, or simply amending the merge commit for other reasons)
    i. Check first whether you could imagine equivalent code that uses auxiliary variables to keep changes from left and right in different lines and statements.
    ii. For example, a statement such as `variableFromRight.methodCallFromRight(new ObjectCreationFromBase().methodCallFromLeft())` (marked as both left and right) is equivalent to `x=new ObjectCreationFromBase().methodCallFromLeft(); variableFromRight.methodCallFromRight(x)`, in which the first statement would be marked as left, and the second would be marked as right, simplifying the analysis.

            iii. Similarly, `variableFromRight.methodCallFromRight(new ObjectCreationFromBase(argumentAddedByLeft))` is equivalent to `x=new ObjectCreationFromBase(argumentAddedByLeft); variableFromRight.methodCallFromRight(x)`

6. Again analyzing the merge commit, check whether there might be a Confluent Flow (CF) from the execution of the changes in left and right to the execution of a common statement that comes from base
    a. We consider that there might be such a flow when the changes (additions and modifications) in each branch might semantically (that is, its execution) involve **writing** operations to different state elements that are transitively associated with **reading** operations executed by a statement that comes from base
    b. Similarly to what explained for DF, CF analysis involves understanding the execution of code, we might also have to analyze the behavior of methods and constructors called by the declaration under analysis. So, for example, the code "`x=0;\\left ... y=0;\\right this.m();`" requires the analysis of the body of `m` to understand if it has a confluence point, that is, a statement that comes from base and reads both `x` and `y`. Contrasting, in the code "`x=0;\\left ... y=0;\\right this.m(x,y);`" we can assert that there is CF because the method call is the confluence point, as both `x` and `y` are read when executing the method call.
    c. To establish if there is a CF, do not think about the current implementation of CF, simply follow the conceptual definition of CF as in the previous items
    d. Please check the other comments made for DF as they might apply here
7. Again analyzing the merge commit, check whether the execution of the changes in left might Override an Assignment (OA) from the execution of the changes in right, or vice-versa
    a. We consider that there might be such a flow when the changes (additions and modifications) in one of the branches might semantically (that is, its execution) involve a **writing** operation to a state element that is also associated with a **writing** operation involved in the changes (additions and modifications) made by the other branch **AND THERE IS NO WRITE OPERATION FROM BASE IN BETWEEN THEM!!!**
    b. To establish if there is OA, do not think about the current implementation of OA, simply follow the conceptual definition of OA as in the previous items
    c. Field references like `o.a` and `o.b` are considered different state elements and can be written by the branches without leading to OA. On the other hand, `o` is also a different state element, but cannot be changed together with `o.a` or `o.b` without causing interference; but they (`o` and `o.a`) can be written by both branches without leading to OA.
    d. CHECK IF THE SAME APPLIES FOR ARRAYS, MATHEUS
    e. Please check the other comments made for DF and CF as they might apply here

8. Finally analyzing the merge commit, check whether there is locally observable interference. There is no need to answer all the questions below; answering just one is enough.
    a. The changes in one of the branches interfere (negatively affect) with the changes in the other branch?
        i. Is there a state element x that left (right) computes a different value than base and merge? or
        ii. left, right and base compute the same value for x, but merge computes a different value?
    b. Does the behavior of the integrated changes (in the merge commit) preserve the intended behavior of the individual changes (in the left and right branches)?
        i. Do the new effects of left (right) in relation to base are preserved in merge, which has no new effects besides the ones in left and right?
    c. Can you think of a specification (pre-post condition pair) that is satisfied by the declaration in left (or right) but is not satisfied by the declaration in the merge? The specification should refer only to the state elements that are transitively involved in the changes in left (or right).
    d. Can you think of a test that would pass in left (or right) but fail in base and merge? The test assertion should refer only to the state elements (which might involve exceptions, as explained earlier) that are transitively involved in the changes in left (or right).
    e. Can you think of a test that would fail in left (or right) but passes in base and merge? The test assertion should refer only to the state elements that are transitively involved in the changes in left (or right).
    f. Field references like `o.a` and `o.b` are considered different state elements and can be changed by the branches without interference; `o` is also a different state element, but cannot be changed together with `o.a` or `o.b` without causing interference. The same applies for `a[0]` and `a[1]` (instead of `o.a` and `o.b`), and `a` (instead of `o`). The underlying reasoning is that `o.a` can be changed without changing or affecting the value stored in `o.b`, whereas when we change `o` we automatically affect the value stored on `o.a`. Contrasting, reading `o`, as in `this.m(o)`, not necessarily means reading `o.a`, `o.b` and all the fields of `o`. Elements of primitive types, strings, files, and streams are considered atomic; that is, changing different bits of the same integer is affecting the same state element, and so is changing different parts of the same text file, or writing to the same stream.
    g. If the changes in at least one of the branches affect a class signature (renaming of the class or its members; addition, removal, or visibility changes of class members), the previous two test related questions should assume that the signatures can be made the same in all commits without affecting behavior.
    h. If the changes in one of the branches are simply whitespace changes, or changes in comments, there is clearly no interference.

i. If the changes in one of the branches are simply structural (extract field, method, class, renaming, etc.) refactorings, and the resulting merge is valid (syntax and static semantics), make sure that they actually do not change behavior, and we can say that there is no interference.
    i. Changing "`if (x==null) return; y=x; if (x!=null) y=x.n();`" into "`if (x==null) return; y=x; y=x.n();`" is a refactoring but is not structural, and so can lead to interference. So we cannot say that there is no interference without analysing the changes in the other branch.
    ii. Changing the interface of a class C is not a refactoring of C. For example, changing method m of C from "`void m(){this.x=1;}`" to "`int m(){this.x=1;return this.x;}`" is not a refactoring of C, as this breaks clients of C. But this change in the declaration of m, together with changes that fix all clients of m, is a refactoring of the whole system if the clients are properly fixed in a way that preserves behavior. So if a class D that calls m as in "`c.m();r=c.x`" is changed to have "`r=c.m()`" then we can say that D was refactored (even if C was not). If D is the class under analysis, we can consider that it was structurally refactored, and so there is no locally observable interference.

j. If a field (instead of a method) declaration was changed in both branches, the analysis is easier: there is interference only if both branches change the initialization part of the declaration with different values. It's equivalent to the two branches changing the same assignment statement in the same method body.
    i. In these cases, there is no need to fill the DF, CF and OA columns, as we don't even consider that the analysis could be called in such situations (the mining framework doesn't invoke the analysis for these scenarios)

k. In many, but not all, situations the possibility of a DF, CF or OA does not imply on locally observable interference for a number of reasons:
    i. the possibility of DF, CF or OA is never realized due, for example, to a condition that is never true;
    ii. the detected DF, CF or OA already existed in base, and appeared in our analysis because the changes in the lines involved in the analysis had only whitespace changes or structural refactorings;
    iii. the detected DF, CF or OA did not exist in base, but has no harming effect (for example, when the changes in left and right assign the same value to the same variable; we have OA, but no actual interference).
    iv. That’s why we have to answer one of the questions above, instead of simply checking if the analysis was positive in the previous items of this list.

l. In many, but not all, situations the lack of DF, CF or OA implies the lack of locally observable interference. This occurs especially because the analyses might not be sound (they might lead to false negatives, as when the code uses reflection, for example), and because interference could be associated with other kinds of analysis that we have not implemented (like an analysis that checks whether changes from one branch affect the control flow of the changes from the other branch).

m. If one of the branches only removes code, then we can easily know that there is no DF, CF, or OA, but there could still be locally observable interference.

9. Check whether there might be a Control Dependence (CD) from the execution of the changes in left to the execution of the changes in right, or vice-versa.
    a. We consider that there might be such a dependence when the changes (additions and modifications, we do not consider the effect of removals on CD) in one of the branches might semantically (that is, its execution) preclude the execution of the changes (additions and modifications) made by the other branch.
        i. When there is a **left branch** in a conditional **CONDITION**, repeat command, or other statements and a **right branch** inside its body, where changes from the left preclude the execution of changes from the right, it can be executed differently and change the data flow, not executing the right changes.
        ii. When the method has more than one exit, it's necessary to analyze whether changes from the left causes a deviation to changes from the right, finishing in an exit and never executing your changes. Each exit is seen as a method finish flow, which can cause the execution flow to be interrupted and not execute the other lines of the method.
        iii. When a statement produces exceptions edges, if the left and right branches are within its scope, all statements can generate an exception and preclude each other's execution, since all statements generate an edge for the finish or change of the flow.

10. Fill the dataset spreadsheet with the information collected in the previous steps