Semantic Merge Conflict Detection

Shikha Mody
smody@ucla.edu
UCLA

Bradley Mont
bmont18@ucla.edu
UCLA

Jivan Gubbi
jcgubbi@ucla.edu
UCLA

Brendon Ng
brendonn8@ucla.edu
UCLA

Abstract

In any modern software organization, parallel workflows are essential for optimal productivity. However, when multiple developers are working on overlapping changes, it introduces the chance of merge conflicts. These can lead to unseen faults and can cost organizations time and money.

The traditional workflow for parallel development is that multiple developers work on separate feature branches, periodically pulling new changes from the main branch into their feature branch. They commit their changes to their feature branch, and when ready, open their feature branch for peer review before merging it into the main branch. However, there are issues with this approach. Its success is dependent on frequent pulls from the main branch into feature branches and an understanding of what effect each incremental update may have on the feature that a developer is working on.

Previous work on merge conflict detection focused mainly on syntactic and textual merge conflicts, broken builds, and broken test cases. We propose to expand this to a broader set of merge conflicts based on semantics. The change in a developer's feature branch should have the same intended consequence regardless of other changes assuming independence.

We propose an automatic semantic merge conflict tool TIM (TIM Improves Merging) to aid in this process. TIM automatically compares new changes to the main branch with your feature branch and notifies you if your change has a semantic conflict. It utilizes symbolic execution to generate an exhaustive list of test cases and is able to identify these silent semantic merge conflicts. Our tool relies on CrossHair, a lightweight Python symbolic execution library. We evaluate this library and it's shortcomings to reason about the performance and behavior of TIM.

CS 230 '22, June 06, 2022, Los Angeles, CA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS Concepts: • **Software Engineering → Merge Conflict Detection**; *Symbolic Execution.*

Keywords: Software engineering, symbolic execution, merge conflict detection

ACM Reference Format:
Shikha Mody, Bradley Mont, Jivan Gubbi, and Brendon Ng. 2022. Semantic Merge Conflict Detection. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (CS 230 '22).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Every engineer who has worked in highly collaborative settings has faced a textual merge conflict when attempting to add their change to the main branch. Two developers edited overlapping segments of code and need to resolve it to incorporate both edits. It is a time consuming process, as one needs to understand the incoming change and update their own changes to incorporate both. Often, it is also necessary to rerun tests to ensure that there were no unforeseen consequences. While this is certainly an issue, one that has been addressed by preemptive notifications, it is not the only potential issue.

Consider the case where a developer pulls code from the main branch and doesn't get a textual merge conflict. This doesn't necessarily mean that the incoming code has not affected the developer's code in some way. This may introduce a breaking change if developers do not write careful test cases and notice behavioral changes of new commits. For instance, if the new commit overwrites a function that the developer uses, it may change behavior while being in a different location in the repository. When incorporating new changes into our feature branch, we need to ensure that there are not these silent, semantic merge conflicts as well.

Our tool, which we call TIM, addresses this issue by automatically detecting these changes through symbolic execution. It compares the feature branch and main branch to see if there are potential issues and notifies users of behavioral changes. It generates an exhaustive, but not necessarily minimal, list of test cases for functions and runs the test cases on each version to ensure that there is behavior preservation. It is based on the equation in Figure 1 for detecting merge conflicts. If the feature branch has different behavior change

$$\Delta[M + B] \neq \Delta[(M + A) + B]$$

Figure 1. Condition for Merge Conflict

based on whether or not a new commit has been added, then there is a semantic merge conflict. TIM is built using the Python library CrossHair and can only reason about Python code. It also has some limitations as discussed in the evaluation section.

2 Related Work

There does exist prior work in the research areas of parallel development and merge conflict detection that TIM takes inspiration from.

2.1 Case Studies Regarding Parallel Development and Merge Conflicts

Prior researchers have conducted case studies which provided valuable insights utilized in the ideation and development of TIM. First off, a group of researchers administered a case study analyzing the change management history of a large, long-established software system which utilized parallel development [6]. They sought to pinpoint how the programmers took advantage of parallel development and the problems associated with it. Specifically, they concluded that parallel development is a ubiquitous aspect of software development that is also highly correlated with errors and quality issues; therefore, it is a crucial concept to study and develop tooling for. This case study assured us that TIM's research area of parallel development was worth further investigation.

Additionally, Oregon State University researchers conducted interviews with software practitioners in order to find out their perspectives when faced with merge conflicts and how this affects their development process [4]. They found a strong correlation between a programmer's perception of a merge conflict and how they choose to solve it, and emphasized how improved tooling can improve programmers' perceptions and overall development workflows as a result. This insight inspired us to create a tool to improve the merge conflict resolution workflow.

2.2 Previous Proactive Merge Conflict Detection Tools

Several existing tools for proactively detecting merge conflicts and notifying developers have played a role in the development of TIM. For example, researchers from the University of Alberta and the University of British Columbia created a tool that examines code branches and utilizes a machine learning classifier to identify "safe" vs unsafe merge scenarios, where safe merge scenarios are unlikely to contain any merge conflicts [5]. After conducting a large-scale study using merge scenarios from GitHub repositories, they reported excellent accuracy in their tool's ability to predict safe and unsafe merge scenarios.

Furthermore, University of Washington and University of Waterloo researchers also conducted a study of merge conflicts in open-source systems. They found that not only are conflicts ubiquitous in the form of text-level differences, but they also appear in the form of build failures and test case failures. They developed a technique known as "speculative analysis," which aims to predict and execute likely next steps taken by a developer, for classifying and mitigating conflicts. Their tool, Crystal, implements speculative analysis and helps developers detect and address merge conflicts [2].

2.3 Symbolic Execution Engines

TIM takes inspiration from the previously stated merge conflict detection tools, but it takes a different approach: symbolic execution. The main reasoning behind this was that many prior merge conflict detection tools assume the existence of an exhaustive and updated testing suite. However, this is not always the case. Although unit and end-to-end tests usually exist in some capacity, it is not guaranteed that they cover every aspect of a large project. As a result, we researched existing symbolic execution engines in the development of TIM. To begin, a tool known as PeerCheck developed by University of California at Santa Cruz researchers is a symbolic model checker for Python [3]. Its implementation, which utilizes the Z3 SMT solver, provides an extremely light-weight approach for symbolic model checking.

The symbolic execution engine used by TIM, known as CrossHair, builds upon PeerCheck, adding support for even more complex inputs such as Python sets and dictionaries [7].

3 Methodology

We created TIM in three separate stages. Since we can utilize CrossHair diffbehavior for refactoring, we have separate tools for refactoring and non-refactoring updates. We also used continuous integration (CI) techniques to automatically run TIM on GitHub branches.

3.1 Theory

As seen in Figure 1, our general motivating idea was to ensure that the behavior change by adding a feature is independent of other changes that have happened since forking a branch from main. Due to CrossHair's ability to work at a function level, we tested function by function to see whether there have been behavior-affecting changes. Our exact methodology depended on whether or not the change was a refactoring change or not. In this case, we take the research definition of refactor, where we expect there to be no behavioral change, only changes to code structure and readability.

3.2 Refactoring Verification

Our work for refactoring was based on one further observation about the equation in Figure 1. In the special case of refactoring, there is an issue with our change is either the left side or the right side of the equation is non-null as well. If B is a refactoring change and we compare the behavior of M to the behavior of M+B we should get that it is exactly the same. If the left side is non-null, we have discovered that our original change was not truly a refactoring change. However, if the right side is non-null, we have discovered that there was some semantic merge conflict that caused our original valid refactor to no longer be behavior preserving.

For refactoring verification, we used CrossHair's diffbehavior tool. Diffbehavior takes in two suspect functions and is able to return an exact test case where they differ. If there is no difference for any test cases, it returns an empty list. This lends very well to the example given above. We first test function M against M+B to ensure that it is null. If not we have an invalid refactor. Then we test M+A against M+A+B to ensure that it is still null. This is the semantic merge conflict detection test. While this tool does not return an exhaustive list of differences, it does not matter since a single difference is enough to say that there has been a mistake.

3.3 Non-refactoring Verification

For non-refactoring verification, we used CrossHair's cover tool. Cover takes as input a function and symbolically executes it to generate a set of test cases that cover all possible execution paths. This addresses the earlier assumption that a thorough test suite exists for large software projects; by generating tests via symbolic execution, we can mitigate potential scenarios from missed test cases. Similarly to the reasoning above, we want to ensure that the behavior for all test cases is the same. First, we generate test cases to compare all the function versions. We run cover on M, M+B, M+A, and finally M+A+B. We take the union of the four sets of test cases and run all functions with this set. We compare the outputs of M with M+B first. The cases where they differ represent the change that we have encoded with our edits. This is done via a set difference. Then we compare M+A to M+A+B. This shows the change after we included the merged change A onto M. If the test cases where our behavior changed and the output are the same between these two tests, then there is no merge conflict. These steps are executed via shell scripts and other generated files to continue the workflow in a templatized manner.

This logic is based on a single function. However, we have to look at all functions in a change set. In order to run this for an entire branch, we run it for all edited functions. We also implemented a higher granularity for CrossHair to run. Rather than comparing function by function, we implemented a mechanism to compare file-by-file. This is what we continue to use for the continuous integration aspect, since merge conflicts are grouped by files.

3.4 Continuous Integration

Once we created our tool, it was important that it ran automatically once new changes were pushed to the main branch. We utilized GitHub workflows to automatically run TIM on code pushes. It logged results to a separate file for each branch to notify users if there has been a merge conflict that needs to be considered. These workflows were implemented with YAML code to sequentially generate test cases, then run them on two different versions of code (one with the updated changes and one with existing code). This allows for a prototype of TIM to run on our repository.

In terms of implementation specifics, the CI pipeline utilized GitHub actions as a framework and ran our code (which utilized CrossHair's cover functionality). Our code contains a variety of intermediate steps in order to generate scripts for all tests to run, the order in which to run them, and also parse any dependencies. The first step of the pipeline is to generate test cases from the main branch's most recent commit. Next, the specific branch in which the push was coming from was evaluated in a similar manner: generate test cases for its most recent commit. Then, these two sets of test cases are combined and run on both the main branch and the individual branch, with their respective outputs being recorded in 'artifacts' of the workflow pipeline. Lastly, these output files are differenced (text diff) to see if they resulted in the same output values; if they did, then there is no semantic merge conflict with the incoming changes. If there are contents in the diff file, then there are semantic merge conflicts.

4 Evaluation and Results

In terms of TIM's performance, we focused on a proof of concept level evaluation. We tested TIM on a small repository where we created different branches with various merge conflicts and tested to ensure that TIM identified them. We found that it correctly automatically detected potential issues when a new change was merged onto the main branch using continuous integration(CI) techniques. This proves that TIM works in finding semantic merge conflicts but does not prove performance or scalability. However, this is a valid evaluation because good development practices dictate that changes should be small and localized to a handful of functions. We can easily run TIM on these small changes even in the situation where we have a large code base. While exact performance metrics have not been generated due to the short nature of this project, it is suspected that TIM would perform well and could utilize this industry trend with minor changes.

In comparison to previous approaches, we are able to find a different type of merge conflict that is more general than test cases failing. This tool provides developers with the

```
def _max_age_to_expires(cookies: dict, now: float) -> _void:
    """
    Translate `max-age` into `expires` for Requests to take it into account.
    HACK/FIXME: <https://github.com/psf/requests/issues/5743>
    """
    for cookie in cookies:
        if 'expires' in cookie:
            continue
        max_age = cookie.get('max-age')
        if max_age and max_age.isdigit():
            cookie['expires'] = now + float(max_age)
```

Figure 2. HTTPie implementation to check for expired cookies[**?**]. Defining a 'dict' type leads to CrossHair crashing because the explicit types for keys/values are not specified.

peace of mind that their change has not been altered by other commits and will not break production environments, which is an expensive mistake.

Since our implementation of TIM is highly dependent on the symbolic execution engine that we selected(CrossHair), it is important that we evaluate the performance of CrossHair as it relates to our tool.

4.1 Prerequisites

To begin, CrossHair itself has a few conditions for it to work well. The first is that its symbolic execution engine must be aware of all functions' parameter and return types in order to generate test cases for them. Although this is a necessity for successful parametrization, it severely limits Python's dynamic typing feature as a language. This carries over to different data structures that may be used. For example, for dictionaries, the explicit types of its keys and values must be defined in order for CrossHair to successfully run. Otherwise, it will crash. This was tested with a real-world open source repository called HTTPie. HTTPie is an HTTP client implemented in Python. One of the helper functions in that repository checks to see if any browser cookies are expired through a dictionary. Initially, if the function defined a dictionary strictly as a 'dict' type, CrossHair would crash because the keys and values could be variable. As a result, modifying the dictionary to have an explicit type with key/value type defined was necessary for CrossHair to run. These prerequisite type definitions would be tedious to add retroactively to a large repository, but serve as good coding practices to enforce earlier on in the development process which allows CrossHair to potentially be leveraged as a tool. For the rest of these examples, we primarily focused on code patterns that are commonly used across projects; by evaluating if the code constructs are successfully processed by CrossHair, we can eventually specialize to different projects.

4.2 Inheritance

CrossHair documentation states that it is able to reason about user defined classes, however it does not do as well with user defined inheritance. In test cases we ran, it was unable to generate test cases regarding polymorphism. In Figure 3 we have two similar implementations of the same basic code. There are two functions, one that accepts the super

```
import dataclasses
@dataclasses.dataclass
class Obj:
    num: int
    def __init__(self, x):
        self.num = x
    def action(self):
        self.num+=2
@dataclasses.dataclass
class Sub(Obj):
    def __init__(self, x):
        self.num = x
    def action(self):
        self.num+=4

def o(a: Obj):
    a.action()
    return a.num

def s(a: Sub):
    a.action()
    return a.num
```

```
import dataclasses
@dataclasses.dataclass
class Obj:
    num: int
    def __init__(self, x):
        self.num = x
    def action(self):
        self.num+=2
@dataclasses.dataclass
class Sub(Obj):
    def __init__(self, x):
        self.num = x
    def action(self):
        self.num+=3

def o(a: Obj):
    a.action()
    return a.num

def s(a: Sub):
    a.action()
    return a.num
```

Figure 3. Inheritance Example

```
def fib2(y: float) -> int:
    y = floor(y)
    if (y == 1):
        return 1
    if (y <= 0):
        return 1
    return fib2(y-2) + fib2(y-1)
```

Figure 4. Recursion Example

object as a parameter and runs an class action, and one that accepts the subtype object as a parameter and runs the overwritten action. There is a slight difference between the two versions, the number we increment by in the overwritten function is different. Due to polymorphism we know that in either versions function o(a: Obj) we could be passed Obj or Sub types. The subtype behavior is different across the two example files however when running CrossHair diffbehavior, it does not find any difference.

CrossHair cover cannot generate test cases with varying user defined types and CrossHair diffbehavior is unable to differentiate between two functions that have parameters with different subclass implementations.

4.3 Recursion and Mutual Recursion

Recursion and mutual recursion have an infinite search space, which makes CrossHair unable to generate more than a single test case despite the numerous possible execution paths that may be taken. While this behavior is logical, it

```
def lambdaFn(a: Callable[[int], int]):
    if a:
        return a(2) + 4
    else:
        return "hello"
```

Figure 5. Lambda Function Example

```
def foo(x: int, y: int):
    if(x > 0):
        return y * 40
    return y * 10

def bar(y: int, x: int):
    if(x > 0):
        return y * 40
    return y * 10
```

Figure 6. Ordering Example

does not test recursion well as there are many possible bugs that may only occur after some number of recursive calls.

In Figure 4, we have a very basic recursive Fibonacci example. CrossHair cover only returns a single test case for this despite it not testing all possible execution paths. This shows it is unable to recursively reason about functions.

4.4 Lambda Functions

For functions that are passed as arguments to other functions, such as callable lambda functions, CrossHair is unable to come up with valid symbolic functions to execute different branches. For example, for code written that has two branches, like in Figure 5, one that is taken when a callable parameter is non-null and one that is taken otherwise, CrossHair only returns one test case(null).

4.5 Renaming and Reordering

CrossHair is unable to figure out renamed functions and reordered parameters without explicit mappings from the user. This is a major limitation since refactoring work often includes renaming functions to more reader friendly names. Also, in an example where we just switch the order of two named parameters (x: int, y: int) -> (y: int, x: int) and maintain behavior of the function itself, CrossHair does not recognize this as a non-behavioral change.

For the code in Figure 6, we ran CrossHair diffbehavior to compare foo and bar. Despite the code bodies being the same and parameter names being the same it reported differences for specific x, y mappings that were incorrect. It shows that with refactors that edit the order of parameters we will have issues even if they are named parameters.

5 Discussion & Future Work

Some areas of future work that would help expand the functionality of TIM are language expansion, new symbolic engines, and evaluation on a larger code base. Currently our dependence on CrossHair inhibits us in a few ways. First, it is only able to reason about Python. While this is a good proof of concept for our idea of Semantic Merge Conflict Detection, it is not a very general use case as only few production level code bases are written in pure Python. Second, as discussed in the evaluation section, CrossHair is unable to reason about a variety of important situations. It is a very lightweight symbolic execution engine which provides performance benefits, but when considering the correctness of changes, it may be better to use a more exhaustive and powerful engine at the cost of execution time. Using a tool that is able to handle object oriented programs would help the adoption of TIM in production. Finally, due to time constraints of the project, we were unable to test our tool on a large code base. Seeing how TIM performs on a code base with thousands of functions and spread out commits would help validate its usefulness.

An alternate approach we could have taken is to do program differencing between the two versions to see what has changed. The main issue with this is it requires manual effort by engineers to evaluate the differences of each new merge into their feature branch. While this would likely perform better in terms of matching functions and identifying specific changes, it still is error prone since humans are likely to overlook edge cases.

5.1 Threats to Validity

As far as internal validity goes, we made the assumption that knowledge and detection of semantic merge conflicts would help reduce production errors and help developers find bugs. However, we did not strongly investigate this claim. It could be the case that developers spend more time handling the notifications and stringent differencing that TIM does than it is worth.

For external validity, our tool is dependent on function level behavior comparisons. It does not expand well to cases where we break a function into two smaller functions, or higher level changes such as with type declarations. In addition to this, the lack of generalization to other high level languages is a problem since we would like to reason about a variety of situations.

Finally, for construct validity, we made the assumption that our small test cases of general coding paradigms such as

recursion, object orientation/polymorphism, complex data types, lambda functions, etc. were sufficient in describing the possible changes that are made in real world applications. It could be the case that we are missing other cases and TIM is unable to find other semantic changes. The test cases we came up with in our example repository may not be reflective of real world merges and may be giving us false positives for TIM's performance.

6 Appendix

Github: https://github.com/shikham-8/CS230-TIM-Improves-Merging

6.1 How To Run Tests

When pushing to a repository with TIM enabled within its CI pipeline, creating a pull request (PR) will automatically run TIM on the changes within that pull request. This is how tests can be run to detect semantic merge conflicts within different code changes.

Acknowledgments

Special thanks to Professor Miryung Kim for her guidance in this project and feedback along the way.

References

[1]]httpie [n. d.]. HTTPie. https://github.com/httpie/httpie

[2] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (5–9). Szeged, Hungary, 168–178. https://doi.org/10.1145/2025113.2025139

[3] Alessandro Bruni, Tim Disney, and Cormac Flanagan. 2011. *A Peer Architecture for Lightweight Symbolic Execution.* https://hoheinzollern.files.wordpress.com/2008/04/seer1.pdf

[4] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. 2017. *Software Practitioner Perspectives on Merge Conflicts and Resolutions.* https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8094445

[5] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. 2019. Predicting Merge Conflicts in Collaborative Software Development. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* 1–11. https://doi.org/10.1109/ESEM.2019.8870173

[6] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. 2001. *Parallel Changes in Large-Scale Software Development: An Observational Case Study.* https://dl.acm.org/doi/pdf/10.1145/383876.383878

[7] Phillip Schanely. 2022. *crosshair Documentation, Release 0.0.23.* https://crosshair.readthedocs.io/_/downloads/en/latest/pdf/