Alva Wei (alvawei)
Jediah Conachan (jediah6)
Kenji Nicholson (kenjilee)
Steven Miller (stevenm62)
Sungmin Rhee (srhee4)

Smerge: Smarter Merge Conflict Resolutions

https://github.com/alvawei/smerge

1 Motivation

Software development often involves collaboration between multiple programmers, resulting in different versions of their project's code throughout development. Version control systems (VCSs) exist to manage these different and potentially conflicting versions. The most common VCS used today is git. A VCS allows multiple developers to make edits to the code independently, and then merge those edits back into the master version of the project. Merging is the process of taking two versions of a file, local (your version) and remote (the version everyone contributes to), and combining them into a single file. In order for the main goal of contributing and collaborating in the same project to be a success, VCSs such as git need to be careful with the merge process. That is, they can't make any changes during a merge that may result in undesired outcomes. For example, if a VCS merged two files and ended up creating a bad interleaving between the two files, that would be undesired behavior and create even more work for the programmer. In the best case scenario, one programmer makes a change to a single file that no one else is working on, and wants to merge their version of the file/project with the remote branch. Git can merge the two versions of the code in this case, and everything works as expected.

Git is not always successful during the merge process, however. We define a merge conflict as what happens when two contributors make changes to the same piece of code and one contributor merges their code with the remote version before the other. When the second person merges their code which interferes with the code that the first person wrote, there is a merge conflict. For example, say person 1 edits an existing variable (existing meaning one that both programmers had in their file before making any changes), say x=2, and changes it to be x=3. Person 1 pushes their code to the repository, and person 2 still sees x=2 in their version of the file. Person 2 changes x=2 to be x=4, and tries pushing their code to the repository. Since person 2 didn't pull the latest changes from the repository before making their changes, git will not automatically merge the updated remote version (that person 2 has not seen yet), with person 2's existing changes. These competing changes could be as harmless as an extra line of whitespace or different variable name. Less harmless merge conflicts will require a resolution that changes how the code works. Here is an example of a less harmless merge conflict:

> Person 1 and person 2 pull from the same version of the remote project and begin writing tests for some similar source code. Say they are making changes in the same file, and person 1 removes a test from the test suite and adds a new one to replace it. Person 2 edits the same test that person 1 deleted, and pushes their code to the

repository. When person 1 gets finished making changes and pushes their code, there is a conflict. This example is what we classify as less harmless, and is something we would not be comfortable with tackling using our tool. This conflict would require manual input from the user to be resolved.

The typical VCS, such as Git, uses line-based analysis to detect merge conflicts. Each line of code is treated as an atomic unit, and a change anywhere in the line is seen as a change to the entire line. This approach often detects “false” conflicts. We define false conflicts as ones that simplistic enough that resolving them should be taken care of automatically. Some examples of false conflicts include things like variable name changes, extra added white space, or putting an if statement around a variable assignment. Changes to the same line of code do not necessarily conflict if the changes are made in two separate regions within the line. What we determine to be a “true” conflict occurs only when the changes overlap in the same region. Current VCSs are unable to make this distinction because they handle a single line as one unit. Whether the conflict be either “true” or “false”, the user is required to manually edit the file that has both changes in it, until the final version is correct. Automatically resolving such “false” conflicts is just one way to reduce the work needed by programmers. Our goal is to reduce the number of conflicts presented to the users and provide a merging module that can be used with several different language parsers (i.e. python, java, etc.). To do this, we will automatically handle as many trivial merge conflicts as possible. Conflerge[4] is a project that attempts to solve the same problem, but we believe it is insufficient in solving the problem for the following reasons:

1. Conflerge’s use of JavaParser imposes limitations on Conflerge. The result of a merge is all formatted according to the default Javaparser formatting. Any custom whitespace, etc is NOT preserved.
2. Since it uses JavaParser other languages are not supported.
3. When there is a failure to merge (meaning that the tool couldn’t resolve the merge conflict automatically), no information is given to the user as to why it failed. Manually merging with git is required at this point. In the case of a failure, we plan to provide the user with a location in the file that caused the merge conflict instead of exiting silently if time permits.

Throughout the report, we will use Conflerge[4]’s evaluation technique to evaluate our tool.

2 Current Approaches

Developers already have a few options to facilitate the merge conflict resolution process. Git has options to ignore whitespace changes[1] while attempting the initial merge, and there are a multitude of merge conflict resolution tools for three-way merging, visual representation of merges, viewing merge history, etc. Some of the more popular merge tools:

- kdiff3[2]
- P4Merge[3]
- diffmerge[4]
- Meld[5]

Most of these existing merge tools are comprehensive, all-in-one merge solutions. Not all of these tools are merge automators; most of them provide an interface for the user to try different methods of merge automation. For example, kdiff3 provides a GUI, a code editor, color-coded difference visualization, and an automatic merge facility with several options for automatic conflict resolution[6], to name just a few of their features. Smerge, on the other hand, provides one specific way to resolve a conflict, AST merging. In the future, we could see Smerge being integrated into an existing merge tool like kdiff3 as an additional option for automatic conflict resolution.

Another approach by previous CSE 403 students, Conflerge[4], handles merging by either parsing code into abstract syntax trees (ASTs) and merging the trees of the conflicting commits or tokenizing the input and diffing the tokens using the Wagner-Fischer algorithm[3]. However, as mentioned above, we believe Conflerge is not an ideal tool, and our solution will attempt to improve on their work.

3 Architecture & Implementation

Smerge currently acts as a git mergetool, allowing developers to use our tool with ease. There are alternatives like Mercurial and Darcs, but we chose git on account of its wide popularity as a VCS. Most developers are accustomed to using git, so incorporating our tool into git leads to an even more convenient process. After a user runs a failing “git merge” command, they may invoke *smerge* through the “git mergetool” command:

```
git mergetool --tool=smerge <conflicting file>
```

This passes the following necessary file locations to our tool:

- $BASE: The original file modified into two conflicting versions, $LOCAL and $REMOTE
- $LOCAL: The conflicting file version the user has modified.
- $REMOTE: The conflicting file version of the branch the user is attempting to merge with.
- $MERGED: The output destination where the final merge is written.

The following diagram (Figure 1) illustrates the high level operations of our tool:

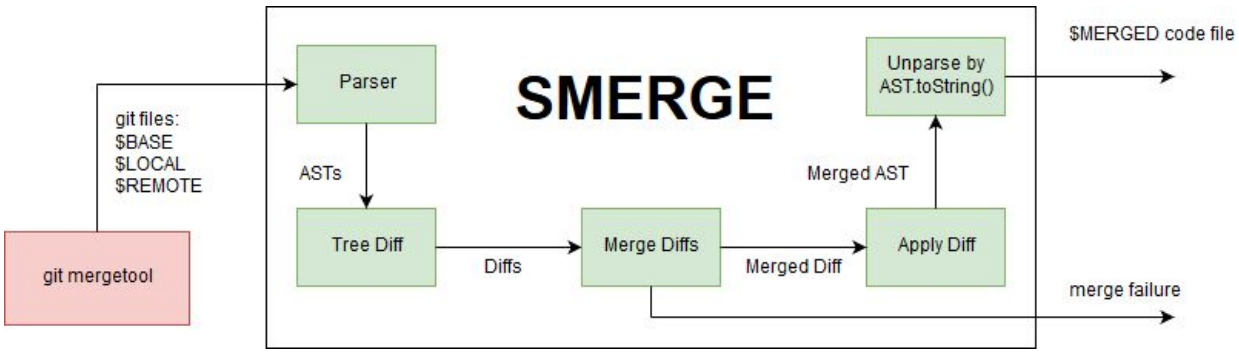

Figure 1

When a user runs the “git mergetool” command shown above, *smerge* will be run with the necessary files passed in as arguments (see Figure 1).These three conflicting file versions (base, remote, and local) are first parsed into their own abstract syntax tree (AST). Next, the base AST is compared to both the remote and local ASTs, producing two different tree diff objects. Each tree diff is defined as a list of actions, where each action represents a change from the source tree (base) to the destination tree (local or remote). These actions include inserting a node, removing a node, moving a node, and modifying a node.

If the two diffs are non-conflicting, then they are merged into a single set of actions (otherwise our tool fails to produce a conflict resolution). These actions are then applied to the base tree, producing a “merged” AST. This final AST must be converted back into a source code file, and is written to the $MERGED file location.

3.1 Existing Tools

3.1.1 JavaParser & Conflerge

JavaParser is an analytics focused tool that converts java source code into ASTs, and is the backbone of our predecessor Conflerge. JavaParser’s AST’s are guaranteed to be unparsed into valid Java files, but are overly complex for our tool’s needs, and do not preserve the exact original source code when parsed such as code clarifying whitespace. As JavaParser works only with Java, we theoretically only need to import Conflerge into our own tool if we wish to use it at all.

One type of merge conflict that Conflerge claims to be successful with is import declarations. Conflerge handles these conflicts by combining the sets of import declarations from the local and remote files, and including the union within the final merge file. We will most likely implement this within our tool as well.

3.1.2 GumTreeDiff

GumTree[5] is described as “a complete framework to deal with source code as trees and compute differences between them,” [5]. In other words, GumTree can parse source code files into ASTs, and then produce a diff between two such trees. As GumTree supports several languages (Java, C, Javascript, Ruby, and more in the future), this tool seems to have a lot of potential. GumTree is, however, heavily dependent on other parsers and does not support unparsing, making the tool extremely difficult to use directly for our purposes. For example, GumTree uses the Eclipse JDT parser for java code, which discards comments when generating an AST.

3.2 Implementation Plan

Due to time constraints, the focus of our project is to create a foundational tool that can be easily built upon. As such, our implementation will be centered on a generic AST that the diffing and merging processes work with. This is so that each supported language only requires a parser that parses a source file into a self-unparsing tree (such that the tree’s toString() method returns the original source content) compliant with the generic AST.

We plan to begin with support for one language, Python, due to it being a simpler language to parse. We need to support at least one language before implementing the tree diffing and merging algorithms. Once our tool has basic functionality working with Python, we will add support for a second language (still to be determined) to ensure that our generic AST and algorithms truly are generic.

Once we have support for two languages, we will improve the parsers and generic algorithms, time permitting.

4 Assessment and Experiments

4.1 Question

To what extent is Smerge successful at resolving conflicts? Does Smerge improve upon previous attempts at automatically resolving conflicts?

4.2 Hypothesis

Smerge will reduce the amount of merge conflicts experienced by the programmer.

4.3 Procedure

To assess the viability of Smerge, we will follow the following steps:

1. Gather many GitHub repositories and their respective historical data
2. From the historical data, look for merge commits that have two or more parents
3. Use git's standard merge tools on the commits to see how many conflicts arise as a baseline
4. Use Smerge's merging algorithm and record metric information
5. Compare the human resolution to the resolution presented by Smerge automatically. If they are the same, report the merge as correct, correct w/o comments, or incorrect.

4.4 Metrics

We begin by defining the true and false positives and negatives relevant to our evaluation:

- **True Positive:** A true positive results when a merge conflict is found, but it requires a manual user resolution to resolve. For example, if two developers change the same line of comments, there is no way to decide which one to keep and which one to discard.
- **False Positive:** A false positive results whenever a merge conflict is found that the tool believes requires manual resolution, but the conflict itself is trivial enough that it does not. For example, if two developers commit the same file, but with different amounts of whitespace, some tools will flag this as a conflict when it should really be resolved automatically.
- **True Negative:** A true negative results whenever the tool detects no merge conflicts and accordingly merges the two commits without error. This is the ideal merging case.
- **False Negative:** A false negative results when there is actually no merge conflicts between the two parent files, but the tool still modified the code as if there was one.

When following the above procedure, the following metrics will be recorded when analyzing a repository. The discussion above on true and false positives and negatives is contained within these metrics and detailed further below:

1. **Conflicts:** The number of merge conflicts (conflicting files, not commits) found in the repository's history with exactly two parents. This does not include conflicts that result from adding or deleting files in the repository.

2. **% Correct:** The percentage of conflicts that Smerge was able to resolve correctly. This number also is the percentage of false-positives encountered, as every conflict solved was one that the standard git mergetool flagged as a false positive.This means completely identical to the human resolution of the code and requires no manual merging.
3. **% Correct w/o Comments/Whitespace:** The percentage of conflicts that were resolved correctly with exception to cases where comments or custom whitespace were modified. This percentage also contributes to the false-positives category.
4. **% Unresolved:** The percentage of conflicts that Smerge aborted because attempting to merge would result in possibly undesired behavior. These conflicts would require manual resolution. This category includes both true positives and false positives, and requires manual checking to categorize the conflicts into the two.
5. **% Incorrect:** The percentage of conflicts that Smerge reported to have merged, but the solution it produced differed from the programmer's manual resolution. This category also contains false-negatives in the case where no changes were necessary but the tool still made some.

4.5 Results

After collecting all of our metrics, we'll place them in the following table. This table also includes the repositories we plan to examine.

Repository	**# Conflicts**	**% Correct**	**% CorrectCW**	**% Unresolved**	**%Incorrect**
TensorFlow Models					
Keras					
flask					
snallygaster					
django					
face_recognition					
ansible					
XX-net					
scikit-lean					
TOTAL:					

4.6 Analysis

After gathering the data, we will analyze it by manually looking at the cases where the tool marked the merge as unresolved. When marked unresolved, this means that the the automation could not find a desired merge. Usually, this occurs when the merge is non-trivial and requires a manual resolution. However, there may be cases where the merge was trivial, but our tool failed to recognize it. From here, the team will analyze each “unresolved” conflict from the top three repos that had the most conflicts and categorize them into the above two categories, placing the results into a table as follows:

Repository	**# Unresolved**	**% T-Pos**	**% F-Pos**
Top Repo #1			
Top Repo #2			
Top Repo #3			

In addition to manually analyzing the unresolved category, we will look at the conflicts that were incorrectly merged to try and categorize some scenarios that caused the merge to fail.

4.7 Conclusions

If our tool manages to resolve a significant percentage of conflicts automatically, then we will have some degree of confidence that our tool fulfilled our goal of handling more conflicts than git’s standard merge tools. However, there are some challenges to this experiment in that it falls victim to both selection bias and undercoverage bias. We only apply our tool to a select few repositories, meaning that we can only infer that our tool was successful. For instance, they may exist some code bases where our tool was no better than git’s standard merge tools because none of the merge conflicts were trivial enough for it to handle.

5 Risks and Challenges

The primary focus of our project will be to provide support for multiple languages. To achieve this goal, we have examined few different open-source parsers available on GitHub, including JavaParser and GumTreeDiff. These parsers have their own definitions of abstract syntax tree that are incompatible with our goals as they remove comments and destroy original source formatting. Lack of documentations in these project make it very difficult to understand their system and build our system around theirs. Consequently, we will build our system from scratch from end to end. Managing time properly will be crucial since we will also have to implement our own parser and AST on top of originally planned merger and unparser. Our hope is that building our own parser and AST will make it easier to build merger and unparser.

As we understand that we may not be able to complete multiple parsers for different languages due to time constraints, our implementation will focus on the generic AST that can represent

different languages, yet can be merged and unparsed using the same algorithms. Our vision is that this generalization will allow our program to be flexible and easily extendable to multiple languages in the future. Different languages have different formatting; therefore they may require the AST to have different properties that are unique to each language. Since we do not have all the parsers already implemented, we need to foresee and envision what these requirements may be, at least for the languages we plan to support if not all languages.

We want to be able to resolve more merge conflicts automatically than other competitor tools. These tools use a well-defined differencing algorithm. To improve their performance, we would have to come up with a new algorithm or tweak the current algorithm to allow more merge conflicts to be resolved automatically. This process can be quite difficult in its own and could introduce more bugs as well. Additionally, we put ourselves at the risk of allowing undesirable merges when we add such flexibility. While we want to minimize the number of false-positive conflicts, it is more important that we do not allow true conflicts to be merged automatically. Automatically merging true conflicts will not only cause erroneous behaviors in the program, but also take away more time from the user to go back and fix the issue.

Week-By-Week Schedule

- Weeks 1-2
    - Choose project, complete project proposal, and begin planning implementation
- Week 3
    - Complete Architecture and Implementation plan and begin implementation
- Weeks 4 - 5:
    - Basic implementation (Parsing code into ASTs, functionality to merge trees, etc.)
- Weeks 5 - 6:
    - Additional implementation (add more features like error output for when merges are unsuccessful to explain why they were unsuccessful. If time permits, add additional conflict handling through algorithms like branch ordering, introduce something new and helpful in handling conflicts)
- Week 7:
    - Begin writing tests and testing our tool, gather data for evaluation by pulling merge history from several popular Git repositories
- Weeks 8 - 10:
    - Finalize project by proofreading specification/documentation, cleaning up code, etc. Then continue testing, start and finish gathering results for evaluation, begin drafting final report
- Week 11:
    - Complete final report

References

[1] Asenov, D., Guenot, B., Müller, P., & Otth, M. (2017, April 29). *Precise Version Control of Trees with Line-based Version Control Systems*[Scholarly project]. Retrieved April 5, 2018, from https://pdfs.semanticscholar.org/cc44/ca01cf8e82437f08374c03d2ef7a63651ec1.pdf

[2] The Git solution for professional teams. (n.d.). Retrieved April 05, 2018, from https://bitbucket.org/

[3] Navarro, G. (2001, March). *A Guided Tour to Approximate String Matching*[Scholarly project]. Retrieved April 5, 2018, from http://repositorio.uchile.cl/bitstream/handle/2250/126168/Navarro_Gonzalo_Guided_tour.pdf

[4] Hanawalt, G., Harrison, J., & Saksena, I. (2017, March 10). *Conflerge: Automatically Resolving Merge Conflicts*[Scholarly project]. Retrieved April 5, 2018, from https://github.com/ishansaksena/Conflerge

[5]Falleri Ean-Rémy, et al. “Fine-Grained and Accurate Source Code Differencing.” *ACM/IEEE International Conference on Automated Software Engineering,* 14th ed., Vasteras, Sweden, pp. 313–324. https://hal.archives-ouvertes.fr/hal-01054552/document

https://github.com/GumTreeDiff/gumtree

We spent an additional 15 hours on this assignment.