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 GitHub (often referred to as 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. In the best case scenario, someone 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. However, a merge conflict is something that happens when two people make changes to the same piece of code and both try to merge their versions with the remote version. 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. Currently, nearly all merge conflicts are manually resolved by the programmer. Since merge conflicts are frequent and often take non-trivial amounts of time to resolve, automating merge conflict resolution will reduce the time and money spent on large projects.

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 that should not require a manual fix. 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 with this project is to make resolving git merges easier and less time consuming. 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, no information is given to the user as to why it failed. Manually merging with git is required at this point. While it is inevitable that merges will fail, we plan to provide the user with more information when our tool is unable to merge automatically.

Throughout the report, we will use Conflerge[4] as a comparison tool, and a way to measure our success.

2 Current Approaches

Developers already have a few options to facilitate the merging process. Git has options to ignore whitespace changes[1], and there are a multitude of merge 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. 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.

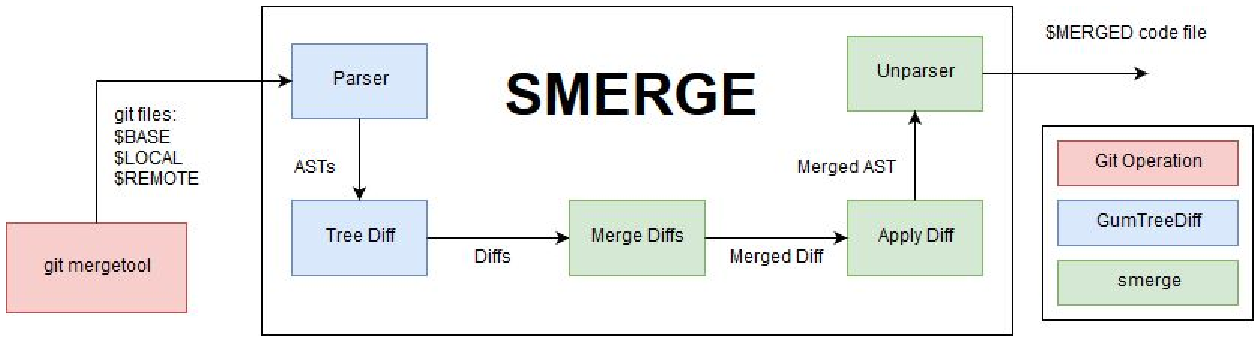
Another approach by previous CSE 403 students, Conflerge, handles merging by either parsing code into abstract syntax trees (ASTs) and merging the trees of the conflicting commits or tokenizing the input and diffing the tokens using the Wagner-Fischer algorithm. However, as mentioned above, we believe 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. After a user runs a failing “git merge” command, they may invoke *smerge* through the “git mergetool” command. 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 illustrates the high level operations of our tool:



The 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 diffs. Each tree diff is a set 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 ASTs 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 getting creating a combined set of import declarations from the local and remote files, and including the entire set within the final merge file. We will most likely include this idea in our implementation.

3.1.2 GumTree

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 for use in smerge. One drawback however is that GumTree does not feature The largest drawback however is that GumTree does not support the unparsing of its trees. Additionally, GumTree seems to lack documentation for non-client use of their code.

3.2 Implementation Plan

Ideally, GumTree can take care of the parsing and tree diffing for us. We will have to implement merging the two diffs produced by GumTree, and applying the result to the original base tree. We would then have to implement an unparser for each supported language to convert the merged tree into a source code file. If we have trouble with this, we may resort to creating our own parsers for each language.

This naturally comes with hesitation. The largest drawback is that we would have to implement most if not all of our codebase from scratch, which is a very large time commitment. However, this would give us a large amount of flexibility in designing a generic AST for the purpose of AST merging (current solutions involve having to work around existing tool’s ASTs). Since every operation would be written towards our tool’s purpose, our tool would also have a greater potential in terms of optimization (for example, our AST would be much simpler than JavaParser’s AST, and in turn quicker).

The job of our parser would be to take in some source code file as input, and then parse it, creating an abstract syntax tree. When the parser is fully functional, we will use it to create ASTs out of the $BASE, $LOCAL, and $REMOTE files provided by git. At this point, we can identify the difference between the ASTs generated from the three files. Wherever the ASTs from $LOCAL and $REMOTE do not match, that means that the two versions are different. If these changes are within the same area of the AST created from the other file, there is a conflict. If there is no conflict, there will be no issues merging automatically. Once we can identify conflicts, our next job is to figure out how to merge the two trees. This will require some sort of categorization into “types” of conflicts, and handle each type separately. Conflicts that don’t fit within one of these types will not be resolved automatically. For things like white space, we could include line numbers in each node, and then identify them as we unparse the code. If time permits, we will have the opportunity to extend this tool to other languages as well.

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 and contrast the human resolution to the conflict with Smerge's merge resolution

4.4 Metrics

When following the above procedure, the following metrics will be recorded when analyzing a repository:

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 means completely identical to the human resolution of the code and requires no manual merging. This percentage also reflects the number of false positives, which means conflicts that were flagged as conflicts, but could be resolved automatically.
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.
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.
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 may also contain false-negatives in the case where merging unintentionally creates them.

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**
androidannotations					
elasticsearch					
fastjson					
glide					
javaparser					
libgdx					
MPAndroidChart					
netty					
RxJava					
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. That said, there are two categories:

1.) **True-Positive**: the merge conflict is a true conflict in that it requires manual action to fix
2.) **False-Positive**: the merge conflict is a false conflict in that it should not take manual action to fix, but is still flagged as a conflict regardless.

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 add support for multiple languages. We need a different parser other than JavaParser that can handle different languages to achieve this goal. We plan to use an open-source tool called GumTree. Much like JavaParser used by Conflerge, it can parse and convert source files into abstract syntax trees. Furthermore, it can compute differences between the trees. This tool claims to support Java, C, JavaScript, and Ruby at the moment. This will hopefully alleviate the burden of having to implement our own parser. One caveat to this approach is that our project will rely heavily on GumTree because it does so much of the work. Conflerge had reported of the limitations imposed by their reliance on JavaParser. Likewise, our project could become constrained by the functionality of GumTree if we were to rely too much on it. Bugs and errors in GumTree could also propagate into our project. Another challenge with using GumTree is that we have to implement merger and unparser based on GumTree API, and GumTree is very poorly documented. It will requires some work to understand and build around GumTree. If GumTree fails, an alternative approach to this would be to implement our own parser and abstract syntax tree. This could be beneficial in that we would have complete understanding of the implementation, and it would be customized for our use. However, we feel that this could be a very difficult task for the given time constraints.

Secondarily, we want to be able to resolve more merge conflicts automatically than what Conflerge and other tools can do. Conflerge uses a well-defined differencing algorithm, and GumTree provides its own differencing tool. 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-R´emy, 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.