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 [8]. 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 are conservative in that when a merge fails, they notify the user that the merge was unsuccessful instead of attempting to automatically solve it. This leads to developers having to spend time to manually edit the files to account for the changes made in the two parent files. 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 able to automatically merge two files, 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. Consider the following simple example in python:

The base version that 2 users edit:

```
1  # Common ancestor (Base):
2  x = 0
3  def doSomething(modify)
4    self.x = 2;
5
```

Contributor 1's commit, note their change in x assignment on line 4

```
1  # Yours (Local):
2  x = 0
3  def doSomething(modify)
4    self.x = 3
5
```

Contributor 2's commit, note their addition of an if statement spanning from lines 4-5

```
# Theirs (Remote)
x = 0
def doSomething(modify)
  if modify
      self.x = 2;

```

In the case of an example like this, git is unable to automatically merge the two commits when both users push their changes. These competing changes could be as harmless as an extra line of whitespace or different variable name, but both will result in a merge conflict that requires manual resolution. We define the notion of “harmful merge conflict” as a conflict which affects the way the code works, such as a merge conflict where there are updates on separate branches on the same exact line. Harmful merge conflicts will require a manual resolution. Here is an example of a harmful 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 harmful, 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 are 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. Additionally, changes to the same line of code do not necessarily conflict if the changes are made in two separate regions within the line. For example, consider a method header with multiple parameters. If one person changes the name of a parameter and another person adds a parameter, those changes occur in the same line of code but shouldn’t necessarily cause a conflict. 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.

It is worth noting that Conflerge[4] is an existing tool that solves many of the above problems, barring merge conflicts that it believes to require manual resolution (edits to the same println statement, for example), but we believe it can be improved upon for the following reasons:

1. Conflerge’s use of JavaParser [6] 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. This is relevant to developers who use custom whitespace in their code for code clarity and readability purposes. For example, custom whitespace is important in industry in that it allows developers to break code into blocks so they may better understand each separate action made in a method, which in turn allows for better code readability. While the conflict may be solved, if a developer cares about custom whitespace, they must manually reinsert it.
2. Conflerge acts as a wrapper around JavaParser, meaning 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. This is a pretty standard practice (can be found in Git), although Conflerge does not provide this feature.

Our goal is to reduce the number of conflicts presented to the users, similar to Conflerge, but also 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 and implement a generic abstract syntax tree that can work with different parsed languages (both of which are detailed later in the report.)

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

2 Current Approaches & Related Work

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; rather, 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.

2.1 Conflerge & JavaParser

An approach by previous CSE 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, for the reasons mentioned above, we believe Conflerge is not an ideal tool, and our solution will attempt to improve on their work.

JavaParser [6] is an analytics focused tool that converts java source code into ASTs, and is the backbone of our predecessor Conflerge[4]. 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, Conflerge also only work with Java projects. However, since our tool aims to be adaptable to several different languages, we decided to create our own generic AST and provide support for individual language parsers to fit that AST. We have written a parser for Python to start with, since it is a simpler language and we wanted to focus more on the implementation of the generic AST merging.

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 included this feature in our implementation as it solved a lot of conflicts for Conflerge.

2.2 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.

Due to the difficulty of unparsing, we decided not to use GumTree in our project. However, we did study GumTree’s diffing algorithm when we were developing our own tree differ.

2.3 Tree Matching and Diffing
Much has been written about tree matching and diffing theory. Some examples:

- “Flexible Tree Matching”, http://theory.stanford.edu/~tim/papers/ijcai11.pdf
- “Abstract Syntax Tree Matching”, http://shodhganga.inflibnet.ac.in/bitstream/10603/36935/12/12_chapter%204.pdf
- “Designing a Tree Diff Algorithm Using Dynamic Programming and A*”, http://thume.ca/2017/06/17/tree-diffing/#the-algorithm
- “Fine-grained and Accurate Source Code Differencing”, https://hal.archives-ouvertes.fr/hal-01054552/document

Most of the sources we looked at did not relate directly to our project’s needs. Some described matching algorithms that were too precise, others described diffing on non-AST-type trees, etc. However, we were able to still able use concepts from these sources to develop our own matching and diffing algorithms to fit our specific needs.

3 Our Approach
Conflerge’s tree merging strategy is so dependent on JavaParser [6] it seems to be almost a wrapper around JavaParser itself. Thus, the primary issue of Conflerge we aim to address is language flexibility. Our solution to this problem is centered on the concept of a generic AST that we can run merging algorithms on. Most (if not all) language parsers generate language dependent ASTs (such as JavaParser), meaning that only one language may be parsed into and unparsed from the AST. Another tool we looked at, GumTreeDiff, could take two source code files, generate an AST for each, and then produce a diff (a set of actions) between the two trees. At first glance, GumTreeDiff seemed to have a potential use to our approach. However, GumTreeDiff, while supportive of multiple languages, is dependent on third-party parsers and their ASTs (such as Eclipse’s JDT Parser). GumTreeDiff also does not support the unparsing of ASTs, and merging trees seems impossible.

Creating our own AST not only grants flexibility in supported languages, but also in the tree merging process (which is explained in more detail in Architecture & Implementation). It also allows us to retain some source code information that other parsers ignore (JavaParser/Conflerge ignore some whitespace, and Eclipse's JDT Parser completely ignores comments). This may make our AST less "abstract" than others, but the extra information is necessary for the tree merging process.

The downside to using our own generic AST is that we must code every other part of our tool from scratch as well. Converting other parser's ASTs into our generic AST seems to be just as much work, if not more, than implementing a parser specific to our AST. The most difficult part of our approach is the tree merging process, which involves multiple steps: node matching, tree diffing, and then tree merging. All of these problems have been solved before, but we must tailor solutions to our generic AST to satisfy our needs. Writing our entire codebase from scratch also allows us to understand how everything works, and leaves room for optimization later.

Thus far, our tool supports only Python 3. We originally planned to support two languages to ensure our AST is language independent, but have decided to spend more time on the tree merging algorithms instead. That said, to support another language only a single parsing class must be created that can parse a source code file into our generic AST format (described in Architecture & Implementation) and unparse an AST (of the same language) back into source code. This parsing class should extend our abstract Parser class. To say that our tool is simply Conflerge but for Python would be incorrect, as our code has been specialized for tree merging and makes it very easy to support other languages, something that is impossible to do with Conflerge.

4 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:

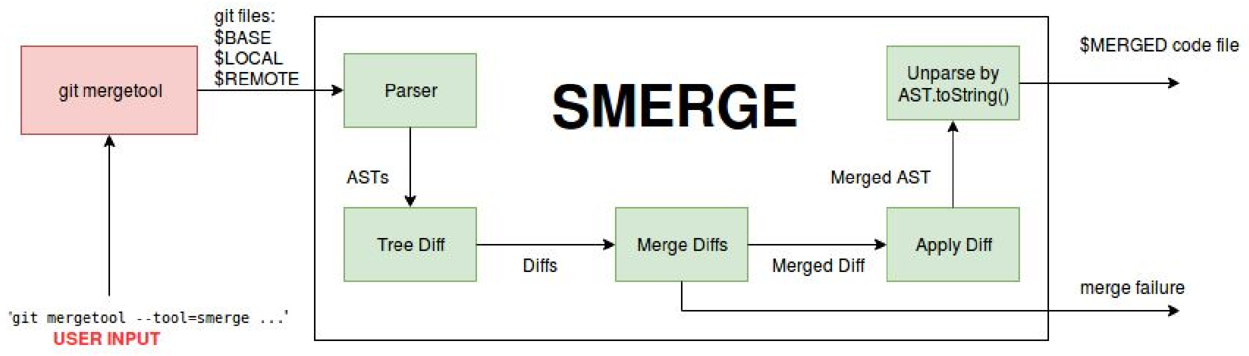

Figure 1

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/updating a node.

If the two diffs are non-conflicting (for example, if one tree moves a node and another one updates it), 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.

If, however, the diffs are conflicting, then the merge fails and Smerge is unable to automatically resolve the conflict. In this case the user must manually resolve it.

4.1 Implementation

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.

Our generic AST is made up of ASTNodes with the following properties:

- The content of the line of code it represents in a string (the "label"). This is used when unparsing the tree back into code.
- Its parent node, which is used for identifying if a node has been moved, for example.
- Its children, as a list of ASTNodes.

- A classification type which is one of the following: root node, import, whitespace, class name, method header, while loop, for loop, assignment, return, comment, or block comment.

Our implementation has 3 main components: a PythonParser, a Matcher, and a Differ.

- PythonParser, as described above, takes in a file and parses it line-by-line into an AST. Note that the PythonParser actually parses the file content into a PythonNode, which must satisfy the requirements of the generic AST nodes as described above. In addition, PythonNodes include information about indentation, which is relevant for Python but may not be applicable to other languages.
- Matcher takes the $BASE, $LOCAL, and $REMOTE AST trees generated by PythonParser and matches nodes by assigning unique IDs to each node. First, each node in the $BASE tree is assigned an ID. Then each node in $LOCAL is compared to each node in $BASE: matching nodes receive the same ID; unique nodes are given new IDs. The same is done with $REMOTE. Nodes are considered "matching" if (1) their types match, and (2) their labels are above a certain similarity threshold to each other. We calculate string similarity by using the Levenshtein distance [7] between two strings
- After Matcher matches the three trees, Differ iterates over the trees to produce two Action sets: one set of Actions that transforms $BASE into $LOCAL, and one that transforms $BASE into $REMOTE. Actions include the following:
    - Insertion: a node that exists in the other tree but not in $BASE
    - Deletion: a node that exists in $BASE but not in the other tree
    - Move: move a node to a different location in the tree - this is syntactic sugar for a deletion, an insertion, and a possible update depending on where it is inserted since the indentation might change.
    - Update: update the node's label

  The two sets of Actions are then merged and performed on $BASE to obtain the final resolution. If merging the Action sets is not possible (e.g. a particular node is updated in one set and deleted in the other), Smerge currently throws an exception. In the future, we will look into ways to handle this differently if time permits.

We began with support for just one language, Python, due to it being a simpler language to parse. One important note: we are currently handling imports separately. If there are differing import statements, we simply put them all together. Since many conflicts are due to missing import statements, doing this solves those right off the bat without having to go through potentially large amounts of code.

4.2 Example

Let's run Smerge on some simple input files:

```
$BASE:              $LOCAL:             $REMOTE:
import A            import C            import B

x = 1               x = 2               if True:
                                            x = 1
```

4.2.1 Parsing and Matching

After parsing and matching, Smerge produces three line-based ASTs with IDs as follows:

```
$BASE:              $LOCAL:             $REMOTE:
(0) @root           (0) @root           (0) @root
  (1) import A        (1) import C        (1) import B
  (2)                 (2)                 (2)
  (3) x = 1           (3) x = 2           (4) if True:
                                            (3) x = 1
```

Note: "import A", "import B", and "import C", and "x = 1" and "x = 2" receive matching IDs. This is because their similarity scores are above the threshold (which will be refined through testing). The "@root" node is just the root node of the AST tree (it does not hold any information about the source code) and is assigned an ID of 0.

4.2.2 Diffing

The Action set from $BASE to $LOCAL:
[Update 1, Update 3]

- Node (1), the import statement, is updated. The text "import A" is changed to "import C".
- Node (3), the assignment of x, is also updated. "x = 1" becomes "x = 2".

The Action set from $BASE to $REMOTE:
[Update 1, Move (Insert 3 under 4[0], Delete 3 from 0[2]), Insert 4 under 0[2]]

- (where x[y] represents the y-th child of the (x) node)
- Node (1), the import statement, is updated.
- Node (3) is moved under the new node (4), which is inserted as the second child of the root node.

The final merged Action set:
[Update 1, Update 3, Move (Insert 3 under 4[0], Delete 3 from 0[2]), Insert 4 under 0[2]]

- Update node (1), the import statement.
- Update the text of node (3), the assignment of x.
- Move node (3) under node (4) and insert node (4) as the second child of the root node.

Note: Even though (1) is updated differently in both trees, we are able to merge it because, as described above, we handle imports by putting them all together.

4.2.3 Merging

The actions are performed on $BASE to achieve a resolution. The resulting tree:

```
(0) @root
  (1) import C
 import A
 import B
  (2)
  (4) if True:
    (3) x = 2
```

4.2.4 Output

The final tree is unparsed by converting each node back to its label (original input files shown for comparison):

```
$MERGED:          $BASE:            $LOCAL:           $REMOTE:
import C          import A          import C          import B
import A
import B          x = 1             x = 2             if True:
                                                          x = 1
if True:
    x = 2
```

This final merged file $MERGED is then outputted to the user.

5 Evaluation and Experiments

5.1 Question

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

5.2 Hypothesis

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

5.3 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 Smerge's merging algorithm on the two parents and record metric information pertaining to whether the tool merged some conflicts or not.
4. Compare the human resolution to the resolution presented by Smerge manually to determine whether its a true or false positive or negative.

5.4 Metrics

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

- **True Positive:** A true positive results when the tool correctly merges the two commits. "Correct" in this case is defined as if it follows the same behavior that the human resolution took, ignoring differences in whitespace and new additions written in the human resolution.
- **False Positive:** A false positive results when the tool incorrectly merges the two commits. This could happen when there are actually no merge conflicts and the tool modified the file, or when there is a merge conflict, but the tool merged the commits incorrectly.
- **True Negative:** A true negative results when a merge conflict is found, but it requires a manual user resolution to resolve. Thus, the tool does not modify the files. 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 Negative:** A false negative 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.

When following the above procedure, the following metrics will be recorded automatically when analyzing a repository. The discussion above on true and false positives and negatives is contained within these metrics and detailed further below. This methodology is a stepping stone to help categorize outputs of our tool into purely positives and negatives, where a "positive" occurs when the tool merged the commits and a "negative" occurs when the tool did not merge the commits.

1. **Conflicts:** The number of merge conflicts (found in conflicting files, not commits) found in the repository's history with exactly two parents. Here, we define a conflict as a portion of the two parent files that conflict. This means that files can contain multiple conflicts, and If multiple conflicts are found between the two parents, all of those conflicts are counted. This does not include conflicts that result from adding or deleting files in the repository.
2. **Modified:** The conflicts that Smerge modified because it deemed the conflict as trivial enough to automatically merge. These conflicts are conflicts that the normal git -merge does not attempt to solve. This contains both true and false positives, and requires manual checking to categorize the conflicts into the two.
3. **Unresolved:** The conflicts that Smerge aborted because it deemed attempting to merge would result in possibly undesired behavior. These conflicts would require manual resolution. This category includes both true and false negatives, and requires manual checking to categorize the conflicts into the two.

After categorizing the conflicts into either modified or unresolved, we manually categorize them as either true or false, where "true" means the tool was correct in its decision and "false" means it was incorrect. Results can be found below:

5.5 Results

As of now, we have only calculated the results for a few repositories, as our tool has not been fully optimized to work with some of the repos [9-15] listed here.

Repository	**#Conflicts**	**#Modified**	**#Unresolved**	**% Modified**	**%Unresolved**
pipenv	45	2	43	4	95
TensorFlow Models	4	1	3	25	75
Keras	135	13	122	9	90
flask	70	10	60	14	85
XX-net	8	2	6	25	75
ansible	196	4	192	2	97
scikit-learn	421	113	308	26	73
TOTAL:	**879**	**145**	**734**	**16**	**84**

Our results for this table can be reproduced by following the instructions described on the README file located in the Smerge [16] version control repository located here: https://github.com/alvawei/smerge/tree/master/scripts

From the data above, we manually categorized the modified files into both true and false positives. Then, we manually categorized the unresolved files into true and false negatives for those repositories. To do this, we would first run the automated scripts above and from those results, we would look at the conflict.py files that it produced and then determine which category each conflict fits into below. More details on the rationale of categorization are included in the Analysis section below. As of now, we've manually checked the conflicts on three repositories that had differing distributions. One with a low percentage of modified files (pipenv), one with a moderate percentage (XX-net), and one with a very few amount of conflicts (TensorFlow Models.) The results are shown below:

Repository	# Conflicts	# T-Pos	# F-Pos	# T-Neg	# F-Neg
pipenv	45	0	2	5	38
TensorFlow Models	4	0	1	3	0
XX-net	8	0	2	2	4

If time permits, we will manually check the rest of the repositories to gather more data.

5.6 Analysis

Based on the three repositories we tested, we noted several causes that contributed to our categorization of the conflicts. In what follows, we have included general behavior that we observed for each true and false positives and negatives category.

True Positives:

- Whenever the merged output from the tool matched similar behavior to the manual merge resolution provided by the repository's developer, we classified the output as a true positive.
- In the case where the developer resolution differed, we looked at whether the developer added new code in the same commit as the resolution and accounted for this by only looking at the fix they provided for the conflict.
- Whenever whitespace was not exact, we counted it as a true positive. In some cases, our tool outputs extra lines of whitespace, causing the two files to be different.
- When merging two parents where the conflict is caused by conflicting variable names for the same variable, the tool chooses the wrong variable name. For example, if one branch named a variable x, and the other changed that variable name to be y, and the

developers' manual resolution keeps the name x, our tool will sometimes choose y instead. We count this scenario as a true positive.

False Positives:

- In some cases, the tool loses pieces of the code during the merge process. This is the worst possible scenario.
- The tool would sometimes break the code by removing a necessary line.

True Negatives

- Some of the merge conflicts analyzed actually required manual resolution: this is the case where our tool decided to categorize it as "Unresolved" correctly.
- Edits to the same line of comments are a true negative because the tool must choose which comments to preserve.
- Edits to the same line of code (i.e. changes to println statements) were also a true negative because the tool

False Negatives

- In some cases, the tool fails to merge conflicts that revolved around single character changes. For example, both the remote and local file would place the same character in the same spot, but the tool detected this as an unresolvable conflict and ultimately aborted merging.

5.7 Conclusions

Currently, our tool is struggling to resolve conflicts in the correct manner, with most of the issues being related to the "false" scenarios described above. However, the tool does indeed solve some conflicts, but it does not report these solved conflicts correctly. We will fix this soon.

Currently, the percentage of conflicts the tool handles correctly is too low to say we have succeeded in our goal. However, considering the low cost to set up and run the tool, it's worth using as a premeasure for the cases it does work in. This low percentage is caused by the tool not handling all of the many possible cases that arise during merging, which is something we underestimated during development planning. In future work, support for these cases of merging actions will help increase the percentage. That said, we plan to work towards achieving positive results.

In all, if in the coming weeks 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.

6 Future Work & Goals

There are many avenues upon which to continue the development of Smerge. Here is a brief and non-exhaustive list of improvements we believe can be made.

- **Output:** We are currently improving this issue through manual testing, but we may not have enough time to solve it completely. While our tool outputs a merged file similar to what a “git -merge” command might output, the actual code outputted may be unexpected to the user, to the point where they’d prefer to use the base, local, or remote file compared to the file our tool gave them. This issue is due to a combination of both bugs and design decisions.
- **Parsers**: Currently, Smerge is only compatible with the Python parser/unparser we developed. In the future, we see support for every language a very realistic goal as all it requires is to parse the language into the format of the generic AST previously described.
- **Advanced Parsers:** Currently, our python parser lacks advanced functionality. For example, it parses method headers as a single node. Ideally, a method header could be parsed into multiple nodes consisting of method parameters, which would allow for more fine tune merging, and thus more conflict resolutions.
- **JUnit Testing:** Currently there is only one small JUnit test that ensures a small Python file can be parsed into an AST, then unparsed back into equivalent source code. Most of our testing throughout our implementation have been manual tests run through main methods, but these are only temporary tests. More JUnit tests would help ensure that our code works as intended.
- **Git Merge Integration:** Currently our tool can only be run by a client as a git mergetool, which was very easy to set up. We feel however that our tool would work better as a git merge strategy, which seems much more difficult and time consuming to integrate into git.

Lessons Learned

For us, design was the core issue we faced throughout the quarter. We struggled to design a tool that contributed something new to the world. When initially choosing this project, we assumed that we would simply use AST merging to resolve some non-trivial projects. However, upon further research we discovered that tools existed to solve these needs. Up until around week 4-5, we were focused on addressing the issue some of these tools had which was the absence of output in the case of a failed attempt to merge. For instance, tools like Conflerge would fail silently without giving the user a reason why. Our resolution was to implement a GUI to communicate portions of the parent files that caused the failure, but it was pointed out by Professor Ernst that this was not a good idea. It was not until the latter half of the quarter that we decided on implementing the Generic AST approach that would solve the problem of tools only supporting one language. This whole segment of indecisiveness taught us that designing a tool that solves a good problem is very hard to conceptualize. Even now, we are still having problems with communicating how our tool is different from previous tools.

Another issue that arose was coming up with a good research methodology to evaluate our tool. Initially, when we were given Conflerge as a predecessor project, we thought that their methodology was solid and could be used in our project. However, after trying to reproduce their results by following their instructions, we ran into compiler errors and failed to obtain their results. This led us to question the validity of their results and taught us the importance of being able to reproduce results. Additionally, we found that using a sound methodology requires hard work. Initially, we thought that we could simply adapt Conflerge's methodology and automatically do all of the evaluation. However, we later found that their assessment is not an interesting assessment. What they did was simply categorize the outputs into lazy categories (similar to what we had before) that could be filled in using bash diffs with the developer's resolution. We found out later that their results seemed very skeptical and unrealistic as the developer's merge resolution could have differing whitespace or have additional additions (such as fixing typos or additional functionality.) Instead of this, we put in the hard work of manually analyzing many conflicts and categorizing them into the reasonable methodology shown in the report. From this, we learned that there is no point in evaluating a tool under a bad methodology; it is much more important to pick a good one as those results are much more interesting and useful for future development.

The last issue we faced dealt with teamwork, with further implications relating to planning and specifications. While working as a team has many positive prospects in that it allows more to get done, we found that it has many negative consequences as well. For one, understanding a teammate's work (which is necessary if one's work builds on that teammate's work) is almost as hard as writing the work itself. This ties into the first implication, which is the issue of specifications. Throughout the quarter, we had trouble understanding the work each of us wrote, and it was reiterated to us that comments play a key factor in a teamwork based project. When we began writing stronger specifications, this process became much easier. Another issue lied in planning, as we initially had a hard time splitting up the work, since most of the work was interrelated and required a lot of communication to get done. We later narrowed it down to 2 of us working on evaluation and 3 of us working on the implementation. This helped our productivity by a lot as it was much easier for 2 or 3 people to communicate with each other than the 5 of us as a whole. From this, we learned that delegating tasks efficiently is vital to exercising the benefits of teamwork.

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.
[6] JavaParser - https://github.com/javaparser/javaparser | https://www.javaparser.org
[7] Rosettacode.org. (2018). *Levenshtein distance* - Rosetta Code. [online] Available at: http://rosettacode.org/wiki/Levenshtein_distance [Accessed 11 May 2018].
[8] Git-scm.com. (2018). *Git*. [online] Available at: https://git-scm.com/ [Accessed 18 May 2018].
[9]TensorFlow Models - https://github.com/tensorflow/models
[10] Keras: Deep Learning for humans - https://github.com/keras-team/keras
[11] Flask - https://github.com/pallets/flask
[12] Docs.pipenv.org. (2018). *Pipenv: Python Dev Workflow for Humans — pipenv 2018.05.18 documentation.* [online] Available at: https://docs.pipenv.org/ [Accessed 25 May 2018].
[13] Ansible - https://github.com/ansible/ansible
[14] XX-Net - https://github.com/XX-net/XX-Net
[15] scikit-lean - https://github.com/scikit-learn/scikit-learn
[16] Smerge - https://github.com/alvawei/smerge

Appendix

Current 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:
    - Continue basic implementation by completing Python parser/unparser, generic AST merging module/diffing algorithm.
    - Begin writing scripts for automatic evaluation
- Week 7:
    - Finish initial implementation of Python parser.
    - Finish initial implementation of AST matcher and differ.
    - Finish writing scripts for automatic evaluation and get some initial results. Debug tool if some repos cause bugs to occur.
- Week 8:
    - Improve matching and diffing algorithms to obtain better testing results.
    - Optimize algorithms to the point where we can run our tool on larger repos.
    - Begin using testing results to fine tune our matching thresholds and diffing preferences (which node to choose when multiple are available).
- Weeks 9 - 10:
    - Finalize project by proofreading specification/documentation, cleaning up code, etc. Tune implementation to work for all intended repositories during evaluation
    - Begin drafting final report
- Week 11:
    - Complete final report and presentation

Previous 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 the GUI. 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

Schedule Changes

In our initial schedule, we severely underestimated how long it would take to finish the basic implementation. We also ran into an issue around Week 4-5 where we scrapped our current plans for implementation in favor of other ones. Namely, we decided against implementing a GUI in favor of integrating our tool within git and providing a merging module that can take in parsers for other languages, thereby making the tool work for multiple languages. We also underestimated how long writing evaluation scripts would take. From this, we learned that coming up with a good idea is very difficult and requires time to consider the pros and cons of them. We also learned that it's okay to change ideas midway as it's much more valuable to pursue an actual improvement than to emulate a weaker version of existing tools. Additionally, we learned that time estimation tends to be severely underestimated--for example, we've been running into unexpected bugs left and right that require refactoring of the code base. To sum up the changes, we pushed back our basic implementation and updated it to be relevant to our goal of creating a tool that can handle different languages. This included details about writing a python parser and the generic AST model.

We spent an additional 20 hours on this assignment.