Abstract: Detecting Semantic Merge Conflicts with Symbolic Execution

Muylaert, Ward; De Roover, Coen

Publication date:
2021

Document Version:
Accepted author manuscript

Abstract: Detecting Semantic Merge Conflicts with Symbolic Execution

Ward Muylaert
ward.muylaert@vub.be
0000-0003-0916-0206

Coen De Roover
coen.de.roover@vub.be
0000-0002-1710-1268

Vrije Universiteit Brussel, Belgium

JPF 2021

1 Introduction

When multiple developers work on the same program, they each have a slightly different version. One developer may be working on a feature, another may be fixing a bug. Version control software (VCS) such as Git, provides a feature called "branches" to help manage these different versions. Eventually these branches merge together again, combining the changes from each branch. In this process, different conflicts can occur: textually, syntactically, and semantically.[1] We focus on semantic conflicts. In this case the merge by a tool like Git completes successfully (no textual conflict) and the program is also syntactically correct. However, the interaction of the various changes leads to unintended behaviour in the merged program. We specifically consider three-way merges. In a three-way merge there are four versions of the program: M, the combined version; A and B, the parents of the merge representing the changes that were present in each branch; and O, the nearest common ancestor of both A and B.

Generalising from conflict freedom definitions found in [3], we state that a merge is conflict free for a certain property P if all of these hold for P:

- If $O_P \neq B_P$, then $B_P = M_P$
- If $O_P \neq A_P$, then $A_P = M_P$
- If $A_P = B_P$, then $A_P = B_P = M_P$

2 Semantic Conflict Detection

We apply this generic definition of conflict freedom to detect semantic merge conflicts. Specifically, we want the property P to describe the behaviour of (parts of) the program. To do so, our technique considers path conditions generated by a symbolic execution engine. For a given three-way merge, the technique applies symbolic execution to versions O, A, B, and M of the program and collects all the path conditions. It then compares path condition across the different versions. Equivalences are tracked and the comparison process results in what we call "cross-version paths" (CVP). One CVP contains some combination of a path condition in O, A, B, and/or M. For example $CVP_1 = \{A_1, M_1\}$, with a path condition in A deemed equivalent to a path condition in M, and $CVP_2 = \{O_2, A_2, B_2\}$, with equivalent path conditions in O, A, and B. For every CVP, our technique checks whether the conflict freedom property holds. In the case of the two examples, CVP_1 is conflict free. No equivalent path condition was found in version O, so $O_1 \neq A_1$, and $A_1 = M_1$. CVP_2 however does create a merge conflict: $O_2 = A_2 = B_2$, but there is no M_2 equivalent to them. In our approach, the CVPs causing a merge conflict are then reported back to the user.

3 Implementation and Evalution

We have implemented a prototype of this approach. As symbolic execution engine, we use Symbolic PathFinder (SPF) [2] with one major change. To get a meaningful comparison when doing our path condition comparison, we need to know when symbolic variables x_O and x_A in versions O and A, respectively, are meant to point to the same variable in the source code. While the SPF already parses the source code information out of the `.class` files,[1] the information is not propagated to the symbolic variables. We adjust symbolic variable creation in order to *do* propagate this information. By doing so, our prototype knows to add $x_O = x_A$ when asking the Z3 constraint solver for a solution to a potential equivalence.

Besides this change, we also reworked `PCParser` to still *parse* the `PathCondition`, but not immediately *post* the parsed result to the constraint solver. This avoids parsing the `PathCondition` several times when comparing across versions. We also implemented `not`, `or`, and `implies` methods to SPF's Z3 interface. `PathConditions` are gathered through the SPF listener interface, so no further modifications to SPF are required.

For the evaluation, we were aiming to analyse real world merges. This proved to be less straightforward than anticipated, due to limitations in our prototype, but also in part due to limitations of the SPF. We are still in the process of getting around this.

4 Conclusion

Merging in software programs can lead to various conflicts. We defined what it means for a merge to not cause a merge conflict in function of some certain property. We used this definition as a basis for an approach to detect semantic merge conflicts by comparing path conditions from execution. We created a prototype implementing this approach on top of the Symbolic PathFinder. We are still in the process of evaluating our prototype to analyse the effectiveness of our approach.

References

[1] Tom Mens. 'A State-of-the-Art Survey on Software Merging'. In: *IEEE Transactions on Software Engineering* 28.5 (May 2002), pp. 449–462.

[2] Corina S. Pasareanu et al. 'Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software'. In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2008.

[3] Marcelo Sousa, Isil Dillig and Shuvendu K. Lahiri. 'Verified Three-Way Program Merge'. In: *PACMPL* 2.OOPSLA (2018), 165:1–165:29. DOI: 10.1145/3276535.

[1] When compiling Java files with the debug flag, some rudimentary source code information is saved into the `.class` file.