Detecting Semantic Merge Conflicts with Variability-Aware Execution

Hung Viet Nguyen
ECpE Department
Iowa State University, USA

My Huu Nguyen
Ho Chi Minh City University of Science, Vietnam

Son Cuu Dang
University of Technology
Sydney, Australia

Christian Kästner
School of Computer Science
Carnegie Mellon University, USA

Tien N. Nguyen
ECpE Department
Iowa State University, USA

ABSTRACT

In collaborative software development, changes made in parallel by multiple developers may conflict. Previous research has shown that conflicts are common and occur as textual conflicts or *semantic conflicts*, which manifest as build or test failures. With many parallel changes, it is desirable to identify conflicts early and pinpoint the (minimum) set of changes involved. However, the costs of identifying semantic conflicts can be high because tests need to be executed on many merge scenarios.

We propose Semex, a novel approach to detect semantic conflicts using variability-aware execution. We encode all parallel changes into a single program, in which if statements guard the alternative code fragments. Then, we run the test cases using variability-aware execution, exploring all possible concrete executions of the combined program with regard to all possible merge scenarios, while exploiting similarities among the executions to speed up the process. Variability-aware execution returns a formula describing all failing merge scenarios. In our preliminary experimental study on seven PHP programs with a total of 50 test cases and 19 semantic conflicts, Semex correctly detected all 19 conflicts.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Reliability

Keywords

Variability-aware execution, semantic merge conflicts, n-way merge

1. INTRODUCTION

In collaborative software development, developers make changes to their local copy of the project files, retrieve changes from team members, and share the changes with them. When two or more developers simultaneously make incompatible changes, a merge conflict arises. Such conflicts put the codebase in an inconsistent state and may delay the project. Due to the fear of merge conflicts, developers tend to postpone merging, and this very behavior can lead to real conflicts at a later time [4, 2].

Previous research has shown that conflicts are common and occur as textual conflicts or *semantic conflicts* (i.e., build and test failures) [4]. Whereas textual conflicts are easy to detect by analyzing changes in overlapping lines of code, semantic conflicts are more difficult to identify and resolve [9], as it requires to compile the code and run the test suite. Brun *et al.* [4] reported that 33 % of the 399 merges that a VCS reported as clean merges actually contained semantic conflicts. Thus, given a set of parallel changes, it is desirable to *identify early the if any change subset contains a semantic conflict*, facilitating in *quick accountability and corrective actions*.

Several researchers have proposed mechanisms to *raise awareness* of parallel changes to detect conflicts already during development [12, 13, 1, 8]. An easy way to detect conflicts is to merge changes and execute the test suite. In this context, *speculative merging* actually performs merges among changes before they are requested by developers to identify conflicts during parallel development [4]. Unfortunately, the costs of speculative merging and other approaches of conflict avoidance and conflict detection [5, 15, 3] becomes increasingly expensive as the number of changes and branches grows because tests need to be executed on more and more potential merge results. As any combination of changes may conflict, the search space for finding a minimal set of conflicting changes is exponential. Even ignoring potential conflicts or fixes among more than two changes (e.g., when a third change resolves a conflict among two conflicting changes), comparing all pairs of changes causes quadratic effort, which is expensive with *branchmania* becoming a common phenomenon in many projects [2]. Delta debugging [16] could identify a 1-minimal solution, but might not find a global minimum.

We propose Semex, a novel approach to detect semantic conflicts using variability-aware execution [10]. Semex exploits the fact that many test executions are similar and rarely affected by the various changes. First, we encode all parallel changes into a single program with variability via n-way merging [6] in which we use a Boolean *patch variable* to represent whether a given change is applied to the original program. Second, we use variability-aware execution to run the test cases. Variability-aware execution explores *all possible concrete executions of the combined program* in parallel with regard to all patch combinations and returns a propositional

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...
http://dx.doi.org/10.1145/2786805.2803208

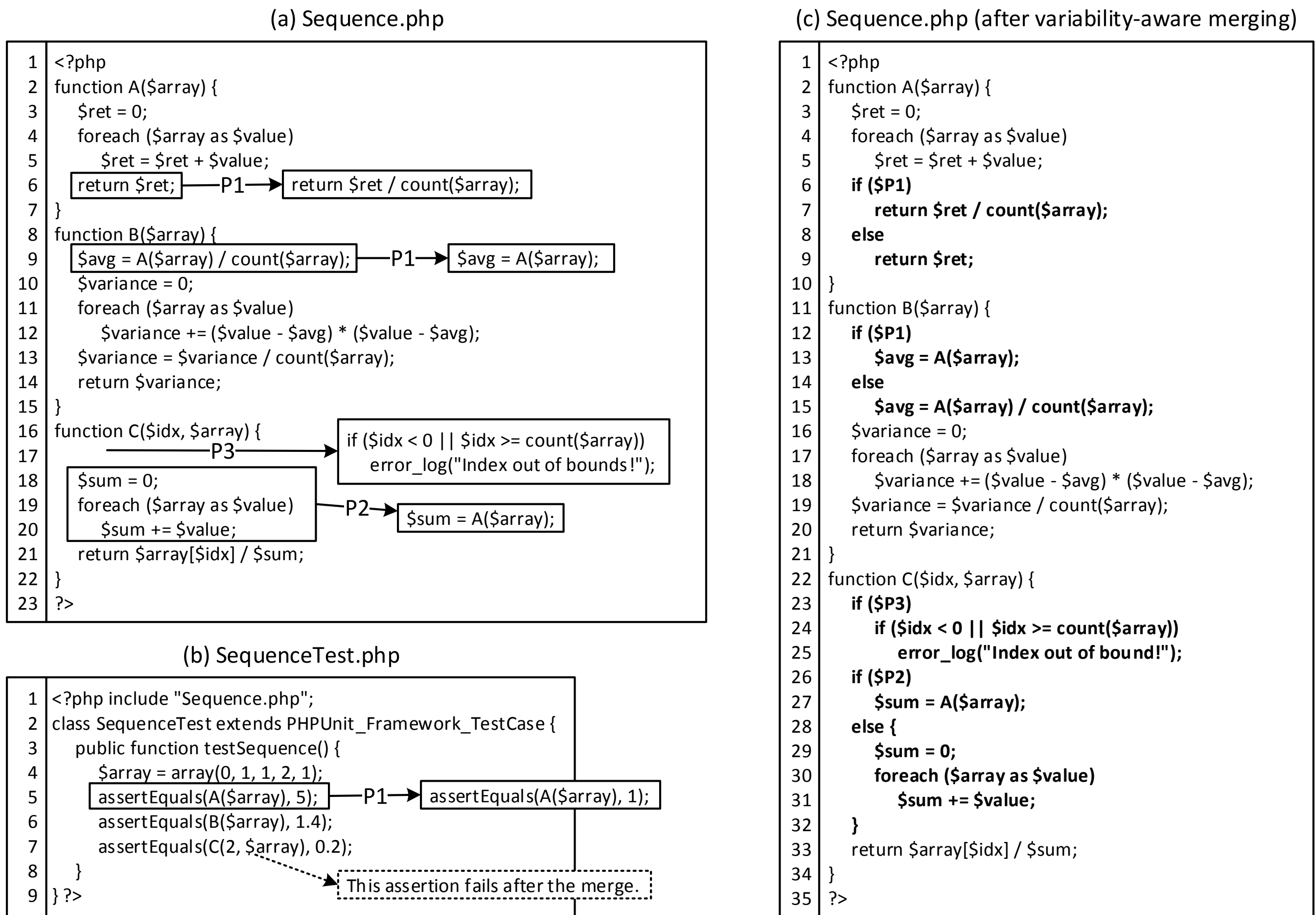

Figure 1: A semantic merge conflict (parts a/b) and the resulting variability-aware merge by our approach in Section 3 (part c)

Table 1: Assertion results on different combinations of the incorporated changes by three developers P1, P2, and P3

P1	P2	P3	C(2, $array)	assertEquals(C(2, $array), 0.2)
Yes	Yes	Yes	1	Fail
Yes	Yes	No	1	Fail
All 6 other cases			0.2	Pass

formula describing *all subsets of those changes for which a test case fails*. The formula describes exactly which changes conflict, including the minimum set. In our preliminary experimental study on seven example PHP applications with a total of 50 test cases and 19 semantic conflicts, Semex correctly detected all 19 conflicts in which the set of conflicting parallel changes reported are the actual (minimum) conflicting changes. In this paper, we contribute:

1. A novel approach to detect semantic merge conflicts of parallel changes using variability-aware execution
2. A preliminary experimental study showing the potential benefit of our approach for detecting semantic conflicts

2. MOTIVATION

Consider a scenario of three developers collaborating on the same simple PHP program in Figure 1a, which computes several statistics for a sequence of numbers. Specifically, function A initially computes the sum of the sequence, function B computes its variance, and function C returns the ratio of a number in the sequence at a given index over the sum of all numbers in the sequence. Suppose that three developers P1, P2, and P3 branch the current copy of the program and make simultaneous changes in their local branches. Developer P1 changes the implementation of method A so that it now returns the average of the numbers in the sequence instead of the sum. Developer P1 also updates a call site of A in method B and its test code (Figure 1b) to reflect this change. At the same time, developer P2 recognizes that part of the code in method C to compute the sum of the sequence has already been provided by method A, without being aware that the implementation of A has been modified by developer P1. Thus, P2 replaces that piece of code with a function call to A. Finally, developer P3 adds an error message for the case that the given array index in method C is out of bounds.

If the three developers committed their changes to the main branch, the changes would be merged without textual conflicts. However, the merged version would contain a semantic error: the assertion on line 7 of Figure 1b would fail if run on the merged version since the code in function C is intended to use the previous version of function A. Note that the test case fails if and only if the changes from developer P1 and P2 are both incorporated into the main branch (see Table 1). Given multiple simultaneous changes, it is nontrivial to identify a set of changes that directly cause a semantic merge conflict since there could exist an exponential number of subsets of the changes.

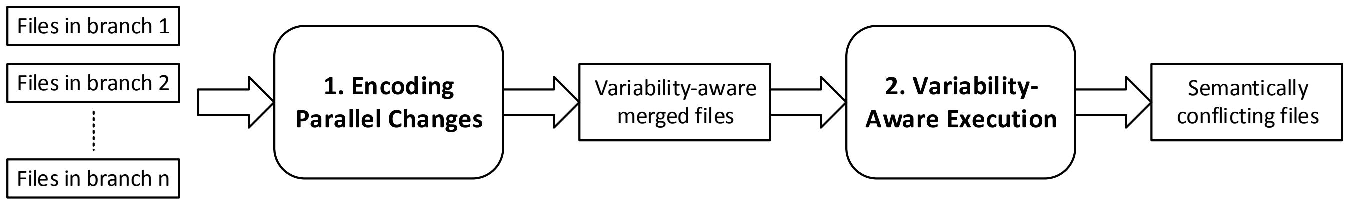

Figure 2: Approach overview

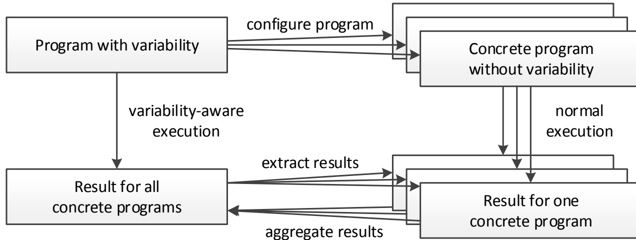

Figure 3: Variability-aware vs. brute-force execution [10]

3. APPROACH

We propose Semex, a novel approach to detect semantic conflicts using variability-aware execution [10] (Figure 2), assuming *textual conflicts* are already resolved. The key ideas are as follows. First, we encode all parallel changes by using special Boolean variables called *patch variables* to *represent whether a given change is applied to the original program*. The result of this step is an encoded program with variability. To illustrate, we show the encoded program for our motivating example in Figure 1c, in which the added if constructs and the patch variables $P1, $P2, and $P3 are used to encode the changes made by the three developers. Then, to identify a semantic conflict, a naive (brute-force) approach could execute the encoded program with all possible values of the patch variables and identify the sets of values that cause a test case to fail. To avoid the combinatorial explosion faced by such a brute-force approach, we instead use variability-aware execution [10] to run the test cases. Our tool explores all possible concrete executions of the encoded program with regard to all possible values of the patch variables and produces results equivalent to brute-force runs on all versions corresponding to all combinations of the changes (Figure 3). When executed on a test case for the encoded program, our variability-aware execution engine returns a propositional formula describing all subsets of those changes for which a test case fails. For instance, in our example, the formula $P1 & $P2 describes all subsets containing changes from P1 and P2, which include the two sets {P1, P2} and {P1, P2, P3}. Such a formula allows us to identify the minimum set of changes that lead to the conflict. Let us detail these steps.

3.1 Encoding Parallel Changes

The goal of this step is to encode simultaneous changes by different developers into a single program with variability. We assume that textual conflicts are detected first by other merging tools. Our algorithm proceeds next as follows. First, we perform *n*-way comparison of *n* parallel versions (made by parallel changes) by incrementally comparing pairs of versions (two-way comparison is widely supported by advanced text-edit tools and version control systems). The result is the alignment of code fragments in parallel versions in which the common code fragments are aligned and code fragments specific to a branch are shown as differences. Second, given the aligned code fragments, we transform the original program into a new program with variability in which the shared code fragments remain intact and the differences are surrounded by if/else constructs with newly introduced patch variables representing the changes made by developers. For example, on line 6 of Figure 1a, developer P1 changes return $ret into return $ret / count($array); we model this change with the corresponding if statement on lines 6–9 of Figure 1c. We encode the changes such that they form valid statements. For a change that cannot be wrapped around with an if statement, we use eval with the change being used as the argument of eval. Note that for simplicity in our example, the patch variables represent the change made by different *developers*. However, we can also use such variables to guard finer-grained changes to detect the conflicts among them.

3.2 Variability-Aware Execution

We next use variability-aware execution [10] to detect semantic conflicts. Variability-aware execution explores all possible concrete runs of the encoded program with regard to all possible values of the patch variables. That is, it efficiently runs all possible versions (of the original program) corresponding to all combinations of changes, taking advantage of the shared code among all versions. When the condition at an if statement is patch variable, it runs both branches and maintains the values of variables in the branches separately. After that, it merges alternative values of the same variable into a Choice representation containing multiple concrete values depending on a condition. Note that the values of potentially many other variables remain unchanged after the if statement is execute, which enables variability-aware execution to scale. To illustrate, let us describe the run of the assertion on line 7 of Figure 1b:

- Function call C(2, $array) is invoked (line 7, Figure 1b).
- In function C, it runs the first if statement with both values (True/False) of the patch variable $P3 (lines 23–25, Figure 1c).
- Next, it explores different branches of the second if statement. It starts by executing the then clause (i.e., calling A($array)) under the condition $P2 == True (line 27, Figure 1c).
- In function A, it executes the for loop and then explores both branches of the if statement at line 6. The returned value is a Choice representation of two alternative values: Choice($P1, 1, 5), i.e., the value is 1 when $P1 == True (the change by developer P1 is incorporated) and 0 otherwise.
- Returning to line 27 of Figure 1c, $sum is assigned with the returned value Choice($P1, 1, 5) under $P2 == True.
- The engine continues to explore the else branch the if statement at lines 29–31 and computes the value of $sum as 5 under condition $P2 == False.
- After running the if statement at lines 26–32, our engine computes the value of $sum as Choice($P2, Choice($P1, 1, 5), 5), which it then simplifies as Choice($P2 & $P1, 1, 5).

- Therefore, the returned value of function C is Choice($P2 & $P1, 1, 0.2) (line 33, Figure 1c).
- Returning to the assertion on line 7 of Figure 1b, the assertion compares the returned value Choice($P2 & $P1, 1, 0.2) with value 0.2 and our engine reports an assertion error under condition $P2 & $P1, which indicates that the test case would fail if those changes from both P1 and P2 were incorporated.

The result of the above execution process successfully identifies a set of changes that directly cause a semantic merge conflict.

Variability-aware execution is similar to dynamic symbolic execution [14] in that they both aim to explore all possible paths in a program. However, variability-aware execution takes advantage of the sharing of variables' values and does not backtrack after executing both branches of a conditional statement. It operates on conditional *concrete* values instead of symbolic values (i.e., a variable may have different values in different conditions, but all values are concrete). Details on variability-aware execution can be found in our prior work [10].

Limitations. Our encoding algorithm currently works for changes to regular program statements (which can be encoded into a valid piece of code within an introduced if statement). We have not addressed the general cases where a change to a field/method declaration is made. In addition, our *n*-way merging algorithm uses a greedy approach based on two-way comparison. Alternatively, we could explore advanced techniques for *n*-way merging [6, 11].

4. PRELIMINARY STUDY

Table 2: Subject programs for evaluation

Example	Test cases	Branches	Semantic Conflicts
MathHelper	7	5	3
Queue1	8	5	3
Queue2	8	4	2
Product	7	4	3
Dog1	4	4	2
Dog2	4	5	2
Chess	12	5	4
Total	**50**		**19**

We performed a preliminary experimental study on seven example PHP applications with a total of 50 test cases and 19 semantic conflicts (Table 2). They are small programs, each with less than 100 lines of code. We constructed these conflicts based on the categorized scenarios of semantic merge conflicts described by Fan and Sun [7]. 14 out of 19 conflicts involve 3 branches; the other involve 2 branches. Then, we manually created the test cases for those programs. Our results showed that Semex was able to correctly detect all 19 conflicts (the set of conflicting parallel changes reported by Semex are the actual minimum conflicting changes).

5. CONCLUSION

In collaborative software development, semantic merge conflicts may occur even when the parallel changes contain no textual conflicts. Therefore, it is desirable to identify early the actual (minimum) set of changes that directly cause a semantic merge conflict. We proposed Semex, a novel approach to detect semantic conflicts using variability-aware execution. First, we encode all parallel changes into a single program with variability in which special Boolean variables represent the presence of the changes. Next, we use variability-aware execution to run test cases on this merged program. Semex then reports the condition indicating the set of minimum changes that cause a conflict. Our preliminary study showed that our approach can achieve promising results.

6. ACKNOWLEDGMENTS

This project is funded in part by National Science Foundation grants: CCF-1318808, CCF-1018600, CNS-1223828, CCF-1349153, CCF-1320578, and CCF-1413927.

7. REFERENCES

[1] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. Fastdash: A visual dashboard for fostering awareness in software teams. CHI '07, pages 1313–1322. ACM, 2007.

[2] C. Bird and T. Zimmermann. Assessing the value of branches with what-if analysis. FSE '12, pages 1–11. ACM, 2012.

[3] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Regression tests to expose change interaction errors. ESEC/FSE 2013, pages 334–344. ACM, 2013.

[4] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. ESEC/FSE '11, pages 168–178. ACM, 2011.

[5] P. Dewan and R. Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. ECSCW'07. Springer Verlag, 2007.

[6] S. Duszynski, J. Knodel, and M. Becker. Analyzing the source code of multiple software variants for reuse potential. In *WCRE'11*, pages 303–307. IEEE, Oct 2011.

[7] H. Fan and C. Sun. Dependency-based automatic locking for semantic conflict prevention in real-time collaborative programming. SAC '12, pages 737–742. ACM, 2012.

[8] R. Hegde and P. Dewan. Connecting programming environments to support ad-hoc collaboration. ASE '08, pages 178–187. IEEE Computer Society, 2008.

[9] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, July 1989.

[10] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. ICSE 2014, pages 907–918. ACM, 2014.

[11] J. Rubin and M. Chechik. N-way model merging. ESEC/FSE 2013, pages 301–311. ACM, 2013.

[12] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantir: Raising awareness among configuration management workspaces. ICSE '03, pages 444–454. IEEE Computer Society, 2003.

[13] A. Sarma, D. Redmiles, and A. van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. FSE'08, pages 113–123. ACM, 2008.

[14] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for C. ESEC/FSE 2005, pages 263–272. ACM, 2005.

[15] J. Wloka, B. Ryder, F. Tip, and X. Ren. Safe-commit analysis to facilitate team software development. ICSE '09, pages 507–517. IEEE Computer Society, 2009.

[16] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.