# Detecting Higher-Order Merge Conflicts in Large Software Projects

Thorsten Wuensche
*SAP SE*
Walldorf, Germany
thorsten.wuensche@sap.com

Artur Andrzejak
*Heidelberg University*
Heidelberg, Germany
artur.andrzejak@informatik.uni-heidelberg.de

Sascha Schwedes
*SAP SE*
Walldorf, Germany
sascha.schwedes@sap.com

***Abstract*—Merge conflicts can occur when multiple developers work concurrently on the same source code corpus. Diverging textual changes (in the same lines of code) are typically easy to resolve with help of common tools like Git.**

**More challenging are *higher-order merge conflicts*. They arise as the result of unintended interactions between changes in different parts of the source code. Higher-order merge conflicts can be caused by a combination of changes, and so even thorough testing of the individual development branches might not be able to identify them.**

**We suggest an approach based on static analysis and a prototypical tool to detect potential higher-order merge conflicts. Our method identifies potentially dangerous dependencies between changed code fragments in a call graph. An evaluation on SAP HANA, a very large industrial product in C++, shows that the approach is able to identify 62% of higher-order merge conflicts causing build failures over 22 months project development time. The same prototype finds no instance of higher-order merge conflicts causing test failures in SAP HANA during a two month development period. In summary, our method scales well and can identify higher-order merge conflicts which escape traditional testing.**

***Index Terms*—version control system, merge conflict, static analysis**

## I. Introduction

Merge conflicts can occur when the contributions of multiple developers to the same software project interact with each other in unintended ways. The most well-known type of merge conflict is caused by changes to the same lines of code. In this case, the contributions cannot be automatically merged and manual intervention from the developers is required. Not all merge conflicts are so easy to find, some may pass the automated merge unnoticed and cause the program to fail its build or show unexpected behavior.

This is referred to as a *semantic* or *higher-order* merge conflict. After merging code changes that were individually harmless, new defects arise in the master branch. As the defects were not present in the development branches, even thorough testing prior to the merge cannot protect the master branch from these conflicts. Higher-order merge conflicts can introduce build failures which prevent the program from compiling. They can also cause tests to fail, or be responsible for unintended behaviour of the code in the master branch.

There are already several tools that detect higher-order merge conflicts such as Crystal [1], WeCode [2] and Behaviour Driven Conflict Identification (BDCI) [3]. These approaches can solve the problem for small and medium sized projects with extensive automated test suits, but they scale poorly for large projects, where compiling the code and executing a test run can take several hours.

We propose an approach to detect potential merge conflicts by identifying code segments that depend on changes in different parallel development branches. We detect the underlying dependencies via static analysis, which avoids the need for costly compilation and test execution steps. By prioritizing speed over accuracy, our method can be used even in projects with a large code base and dozens of parallel development branches. The source code of our prototype is available on GitHub [4].

Our contributions are as follows:

- We develop an approach and a prototype to find potential higher-order merge conflicts. Our approach uses a statically constructed call graph which reuses data from previous runs to scale well with very large source code repositories.
- We evaluate our method on known build conflicts in SAP HANA, a very large software project in C++ and examine the properties and root causes of those conflicts. We also inspect SAP HANA for test conflicts.

This paper has following structure. We introduce the SAP HANA project, its testing process and a taxonomy of merge conflicts in Section II. In Section III we present our approach, and we evaluate it in Section IV. Section V discusses related work. Finally, we state our conclusions in Section VI.

## II. Background and Motivation

### A. Taxonomy of Merge Conflicts

We extend the merge conflict taxonomy of Brun, Holmes, Ernst and Notkin [5] by including possible causes of each type of conflict from the perspective of dependency analysis. We differentiate between textual and higher-order merge conflicts. The latter are then differentiated based on their impact.

*Textual conflicts* are caused by concurrent modifications to the same parts of the source code. The corresponding merges are typically rejected by a version control system (VCS), but it requires manual effort to resolve the conflicts before the merge is accepted into the repository.

Merge conflicts not found by a VCS are called *higher-order merge conflicts*. They can be further separated into multiple categories according to their effect. *Build conflicts* cause errors at compile time, and *test conflicts* result in failing tests. Further categories related to performance or code style are possible, but have not yet been studied in depth.

Note that proximity of concurrent modifications is an important factor distinguishing the two main types of merge conflicts: while textual conflicts can only be caused by modifications in close proximity to each other, code modifications responsible for build conflicts can be also ”far apart”.

### *B. SAP Hana and its Testing Process*

SAP HANA is a database platform developed by SAP which is used as the basis for SAP's business applications such as the Enterprise Resource Planning (ERP) software S/4 HANA. It combines transactional, analytical and spatial data processing as well as text analysis in an in-memory store and offers access to the data via SQL and domain-specific languages. Further properties are a strong focus on multi-core processing and the ability to run on multiple hosts in a scale-out setup.

The source code of SAP HANA consists of more than 30 Million source lines of code which mostly reside in a single large Git repository. Several hundred developers work on the product and produce about 800 software changes on a regular working day.

Testing of SAP HANA is done in a staged approach as shown in Figure 1. Software changes are at first tested locally on the developer's workstation. If the change is considered fit, the developer will push it to the shared Git repository. Before it is actually merged into the shared repository, a snapshot of the current content of the repository is taken into a *shadow repository*, the developer's change is applied to that snapshot and a build and test step is run against that change-specific shadow repository. Only if the build and test step in the shadows repository succeeds, the change is applied to the shared repository and thus made globally visible to all developers. This testing stage is called *pre-submit testing*.

Before SAP HANA is made available to customers, testing continues against the content of the shared repository at stages with increasing resource demand. These two last stages shown in Figure 1 are referred to as *post-submit testing* and *release qualification*.

Pre-submit testing takes a considerable amount of time per software change, often several hours. While the pre-submit testing for a specific change is performed, the content of the change-specific shadow repository ages with every other software change that successfully completes its pre-submit stage and is applied to the shared repository. In other words, a snapshot for a shadow repository taken several hours ago may show a significant deviation compared to the current content in the shared repository. Thus, the result of the pre-submit testing may become invalid.

This is exactly the case when changes have been applied to the shared repository that conflict with a change currently under pre-submit testing. Textual conflicts are reliably detected at patch-application time by existing tools (e.g. Git) that will indicate a merge conflict automatically. However, higher-order

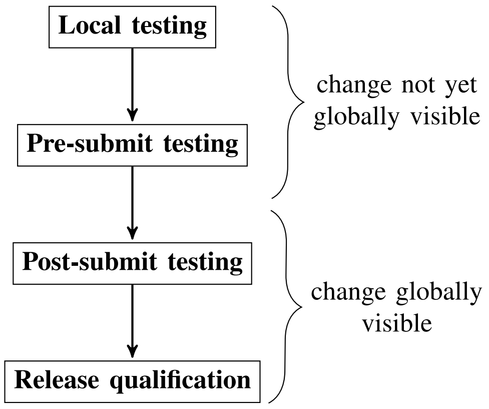


Fig. 1. Testing of SAP HANA during the development process (simplified) and software change visibility.

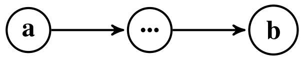


Fig. 2. The changed entity *a* directly or indirectly depends on the changed entity *b*.

merge conflicts may be introduced into the shared repository unnoticed and only be detected at the next build or test run. Such higher-order merge conflicts reduce the effectiveness of the pre-submit testing stage, whose purpose is to avoid the introduction of faulty changes into the shared repository and keep the repository sane.

### *C. Build Conflicts and their Causes*

In our SAP HANA testing practice we observed three main causes for build conflicts:

- Changes to the signature, including modified names, arguments, return values, or even a complete removal of the entity. This occurs when exposed properties of a source code entity are modified and then merged with a changeset which expects the old signature. In this case, the changes expecting the previous behavior have to call the changed entity either directly or through a sequence of calls as shown in Figure 2.
- Missing include statements can break the build. This might occur for instance when one changeset cleans up unused includes, while another begins to use code from the included file, or when source code is moved from one file to another.
- Duplicate definitions will cause build conflicts. They are introduced when entities of the same name are created and not caught by the VCS, e.g. because the definition is placed in different parts of the same file.

All three cases have in common that the two conflicting changes can be located in different parts of the source code, making them difficult to find without compiling. Once the merged code is compiled, the conflict becomes visible by breaking all subsequent builds. This type of conflict is caught by pre-submit testing, as a failing build will cause one of the

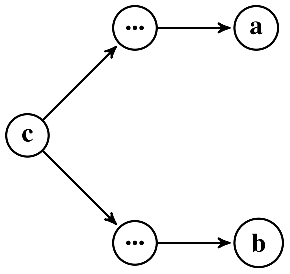


Fig. 3. The changed entities *a* and *b* are called directly or indirectly by the unchanged entity *c*. A test conflicts can occur in *c*.

conflicting changes to be rejected. This assumes that the pre-submit tests of all merges are tested in sequence. Parallel test execution (typically used to speed up the testing process when very long build and test times are expected) can cause presence of build conflicts between merges not tested sequentially.

### D. *Test Conflicts*

Test conflicts occur when the behavior of a code entity was modified while a concurrent change expected the old behavior. Such conflicts can only occur if neither textual nor build conflicts are present, otherwise no tests can be run on the merged code.

Typical code changes responsible for test conflicts found in testing of SAP HANA include splitting a large entity into smaller parts while one part retains the original name (without the full functionality). Other causes are modifications of two separate entities with one of them calling the other. Even more complex situation occurs if unchanged parts of the code call two separate changed entities, see Figure 3.

In general, test conflicts are difficult to find, and they can be responsible for unexpected behavior of any severity. As this conflict type is only defined by its effect rather than its cause, incompatible changes to the product and the test code could also lead to test failures, even though the product's functionality remains unaffected.

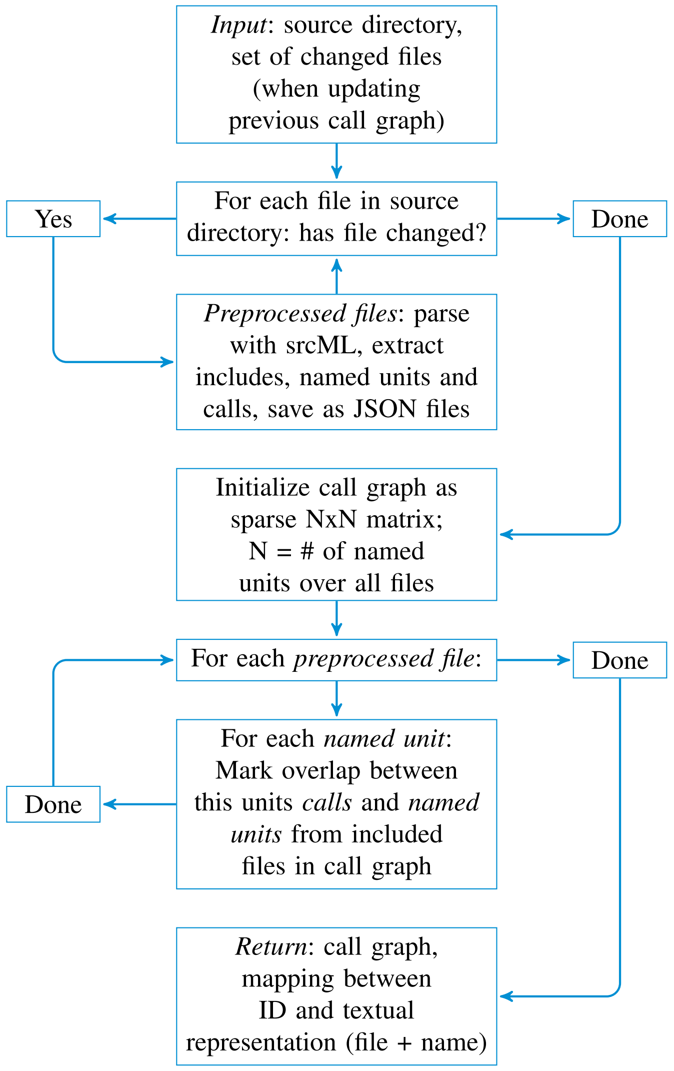
## III. Detection of Higher-Order Merge Conflicts

To detect higher-order merge conflicts we search for potential conflicts between changes that are undergoing pre-submit testing. To this end we construct a static call graph (not considering inheritance and dynamic call dispatching) and search for dependencies between changed code entities.

### A. *Overview of Call Graph Construction*

Figure 4 outlines the call graph construction. Essentially, it consists of two phases. In phase one (top three steps in Figure 4), the source code is parsed and scanned for named units (essentially any named source code elements) and calling units (essentially method calls). The second phase is the core part of graph construction where calling units and named units are matched for changed C++ files (see bottom five steps in Figure 4).

The named units and calling units are characterized as follows:



*Input*: source directory, set of changed files (when updating previous call graph)
For each file in source directory: has file changed?
Yes
Done
*Preprocessed files*: parse with srcML, extract includes, named units and calls, save as JSON files
Initialize call graph as sparse NxN matrix; N = # of named units over all files
For each *preprocessed file*:
Done
For each *named unit*: Mark overlap between this units *calls* and *named units* from included files in call graph
Done
*Return*: call graph, mapping between ID and textual representation (file + name)

Fig. 4. Construction of the call graph for the source code in a given directory. All changed code is scanned by srcML, the result is parsed to extract included files, named units and calling units. Connections are detected via second iteration over all named units.

**Named units** can be classes, functions, structs, constructors, destructors, enums, typedefs, unions, macros or variables. They have an assigned name which can be used to reference them from different parts of the code (including other files). All named units can reference their own name. This ensures that declarations and definitions can be linked together, even if they are located in different files.

**Calling units** are method calls or data types. They potentially contain a call to a named unit.

### B. *Implementation of Call Graph Construction*

The input of our prototypical tool is: (i) path to the source directory, (ii) reference to the master revision (which serves as the basis of the call graph), (iii) references to the relevant repository branches. Each branch is referenced as a pair of hashes representing the branch and the revision of the master branch at the time the merge was queued. As the requested merge stays in the queue while other merges are tested, the revision of the master branch used to test the changes may be more recent.

Our prototype first attempts an *octopus merge* (i.e. a merge with more than two parents) which merges all branches into the master revision at once. If textual conflicts are detected at this stage, the merge fails. For reasons stated in Section II such conflicts need to be resolved first before search for higher-order merge conflicts can continue. We then identify as follows the files which need to be parsed to update the call graph: these are the files which differ between the octopus merge and the master branch revision from the last run (which is recorded in a separate file).

In the next step we use srcML [6] to parse all changed C++ files. The results are scanned for include statements, named units and calls using XPath expressions. Calling units are assigned to the named units that contain them. The extracted information is saved as a collection of JSON files, replicating the original folder structure of the source directory.

After identifying named units and calling units in the changed C++ files, the core part of the call graph construction is performed: iterating over every named unit in every file, and comparing the recorded calling units to the named units of the included files (bottom half of Figure 4). If a named unit matching a calling unit is found, the corresponding entry in the call graph is marked. Sometimes (especially when the unit name is common) several potentially called units can be found. As identifying the correct one is complex and time-consuming, we add all of these connections to the graph.

The call graph construction algorithm is implemented in Python and represents the graph as a sparse Python/SciPy $NxN$ adjacency matrix, where $N$ is the number of named units over all files. The algorithm returns as output the adjacency matrix (serialized as an object Python/NumPy *npz* file), and a mapping from ID to named unit as a JSON file.

*1) Performance Aspects:* The level of transitive includes can be adjusted to find more connections at the risk of introducing false positives. This has significant impact on the runtime. Without transitive includes, the graph can be created in under two minutes, one level requires around twelve minutes. The call graph is recreated during each execution.

If no previous runs are recorded, or the preprocessed files are destroyed, the entire source directory has to be parsed, which takes around two hours for SAP HANA.

Significant runtime improvements can be achieved by skipping the unchanged files. Performing one scan per day reduces the total runtime to approximately 15 to 30 minutes (assuming one level of transitive includes) for all subsequent executions, of which only around two minutes are required for parsing.

### *C. Detection of Potential Conflicts*

Once the call graph is constructed, potential conflicts are detected using the patterns discussed in Section II-C and in Section II-D.

Specifically, we search for three different cases in which two changes can be in conflict:

1) Both changes affect the same named unit.
2) There is a path of calls from one changed unit to the other (Figure 2).
3) There are two paths of calls originating from the same (unchanged) named unit and each of them reaches one of the changed units (Figure 3).

In the following we consider a generalized potential conflict consisting of two changed units (changed by different merges) and one affected unit, with a path of calls from the affected to each of the changes. In the second case, one of the paths contains only the affected unit itself. This means that one of the changed units is also the affected unit. The first case involves only a single named unit, playing the role of the affected and both changed units. Both paths contain only the unit itself.

The workflow used to find these potential conflicts and report the names of the affected unit, the changed units and all units on the two call paths is shown in Figure 5. To facilitate identifying named units which can call two specific changed units we traverse the call graph in the opposite direction.

## IV. Evaluation and Results

We investigate records of build conflicts in SAP HANA development recorded during a 22 month time-span. Where available, we examined the changesets that caused the conflict as well as the commit that resolved it to answer these research questions: RQ1 What type of code changes cause build conflicts? RQ2 How sensitive is our prototype to configuration parameters? RQ3 How common are test conflicts?

### *A. RQ1 What type of code change causes build conflicts?*

Pre-submit testing should ensure that the master branch compiles correctly at all times. Whenever this is not the case, build conflicts are a likely cause. By recording such incidents and the commits that introduced and fixed the build failure, we can examine 54 build conflicts in SAP HANA in 22 months. To identify their cause we search for the two conflicting commits as well as the commit that resolved the conflict. We consider four cases:

Signature changed
: The conflict occurred because the name, arguments or return type of a source code entity were changed or because the the entity was removed entirely.

Include
: An include statement was missing or an included file was deleted.

Duplicate definition
: A variable or function identifier was defined multiple times.

Unknown
: The documentation for many build conflicts was incomplete, missing one of the conflicting merges or the fix. While causes for these conflicts could not be identified, they are still listed to correctly reflect the overall frequency of build conflicts.

Table I shows the distribution of the build conflicts for the considered root causes. In some cases it was either unclear

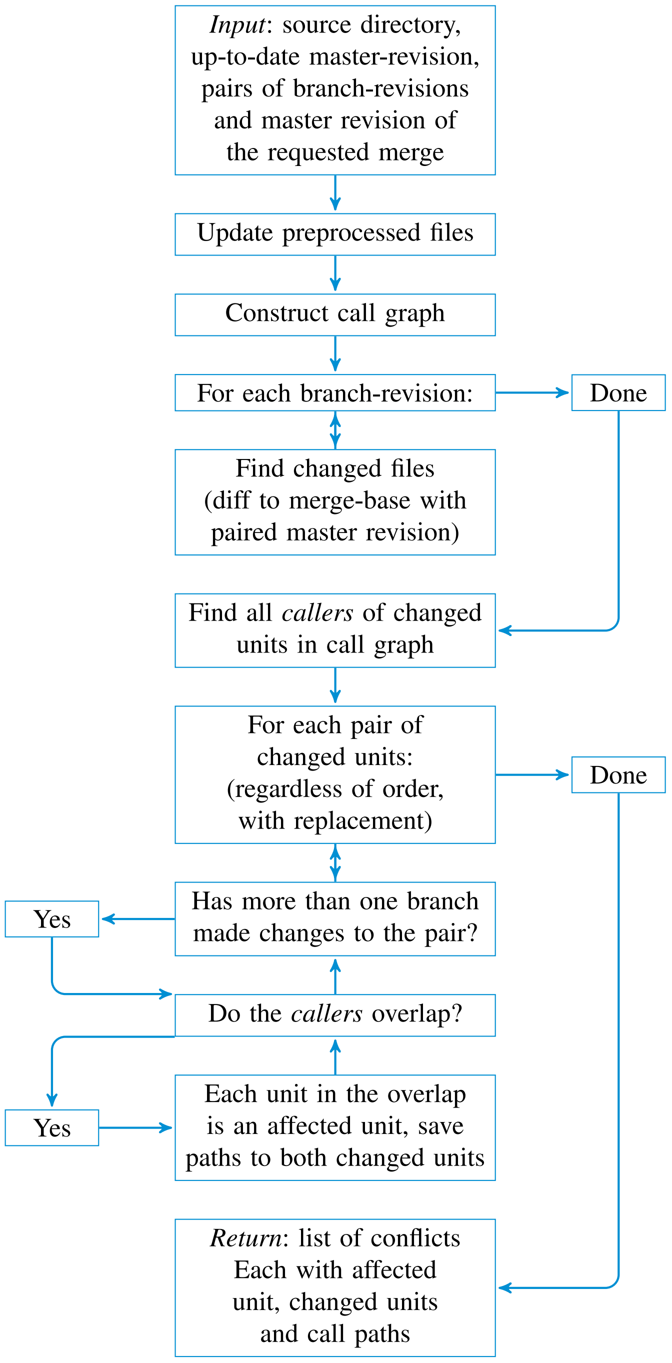

Input: source directory, up-to-date master-revision, pairs of branch-revisions and master revision of the requested merge
Update preprocessed files
Construct call graph
For each branch-revision:
Done
Find changed files (diff to merge-base with paired master revision)
Find all callers of changed units in call graph
For each pair of changed units: (regardless of order, with replacement)
Done
Has more than one branch made changes to the pair?
Yes
Do the callers overlap?
Yes
Each unit in the overlap is an affected unit, save paths to both changed units
Return: list of conflicts Each with affected unit, changed units and call paths

Fig. 5. Complete workflow for finding higher order merge conflicts. After the call graph is created, all paths through it originating from the changed units are tested pairwise for overlaps. If a pair was edited by more than one branch revision and their paths overlap, all named units in the overlap are potentially affected by higher-order merge conflicts.

TABLE I
BUILD CONFLICT CAUSES

| **Cause of conflict** | **Number of Conflicts (out of 54)** | **Percentage** |
|---|---|---|
| Signature changed | 27 | 50% |
| Include missing | 4 | 7% |
| Duplicate definition | 3 | 6% |
| Unknown | 20 | 37% |

which changesets caused a failure or the data was not recorded completely. Therefore, 20 cases could not be reliably assigned to one of the categories and are listed as unknown. They are reported here nonetheless to correctly represent the frequency of build conflicts.

The most common root cause for build conflicts is a signature change (50%). Of these 27 conflicts, 16 were due to a changed or deleted name, which was still referenced in one of the conflicting changesets. The remaining 11 cases were the result of changed arguments or return types.

Less common were missing or wrong include statements (four cases). These were usually due one of two kinds of refactoring: Most commonly, unused import statements were removed, while a conflicting changeset started to use them. There was also a case where a large file was split into smaller parts, causing a file changed in another branch to miss the moved source code entities.

Finally, there were three build conflicts caused by duplicate definitions. In these cases, the duplicated elements did not only share their names, which would be enough to break the compilation, but were completely identical. The VCS did not recognize this duplication due to different positions in the source code, leading to two identical sections in the same file.

Both signature changes and faulty include statements occur as part of refactoring and involve automated changes to large parts of the source code. When for instance the name of a function is changed, many IDEs will highlight calls to that function for the developer, or offer to alter all occurrences elsewhere in the code. With the support of the IDE, it is possible to keep track of all the parts of the source code that need to be edited. This support does not extend to other branches or local copies on the machines of other developers. There, the changes required by the refactoring are missed and can cause build conflicts once merged. This mechanisms might explain a recent empirical findings indicating that the number of merge conflicts increase when IDEs with refactoring support are used [7].

Figure 6 shows the number of C++ files involved in merges that caused build conflicts. As each conflict involves two merges, we list them separately grouped by size and compare them to the number of C++ files in an average merge. 61% of the average merges contain ten or fewer files, and the frequency falls further the more the size increases. The smaller of the chagesets involved in build conflicts behave very similar, with slightly more files on average. The larger of the conflicting changesets however involves over 50 files in 79% of cases. These files are often changed by the IDE as part of a refactoring, which was unaware of the changes made

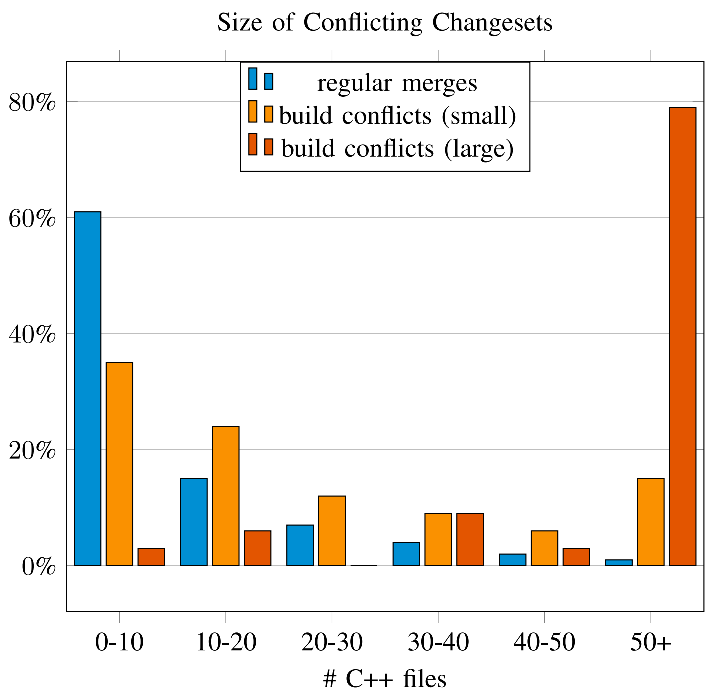


Fig. 6. Number of changed C++ files of merges involved in a build conflict relative to regular merges.

in the second changeset.

*B. RQ2: How sensitive is our prototype to configuration parameters?*

Our prototype depends on two parameters: the distance of transitive includes used in the creation of the call graph and the length of call chains used during conflict detection.

Transitive includes are files not included directly, but via a different include. During compilation, all includes are considered, regardless of how far removed they are, but this requires linking to ensure the correct handling of recurring names. Without a linker, we need to find a balance to ensure potential conflicts can be found without introducing a large amount of false positives.

The length of call chains refers to the number of nodes in the call graph between a conflict and its conflicting changes. When allowing for long chains, the dependencies of all changes overlap at some point, leading to many false positives. Limiting the length too low will cause us to miss conflicts.

The challenge when looking for suitable parameters is reducing the number of false positives to a point where manual examination is possible. Eliminating false alarms completely would require the code to be compiled and test to be run, which defeats the purpose of a fast prediction method.

Table II shows the shortest distances of transitive includes, that enabled our prototype to correctly find a build conflict. 62% of the successfully identified build conflicts conflicts were found when only direct includes were considered during call graph creation. The remaining 38% required indirect includes of distance one, while also maintaining all of the conflicts

TABLE II
PROTOTYPE: TRANSITIVE INCLUDE DISTANCE FOR BUILD CONFLICTS

| Distance | number of conflicts (out of 20) | Percentage |
|---|---|---|
| 0 | 13 | 62% |
| 1 | 8 | 38% |

TABLE III
CALL GRAPH PROPERTIES BY TRANSITIVE INCLUDE DISTANCE

| Distance of transitive includes | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| # connections | 3.511313 | 8.655.580 | 15.092.025 | 19.616.230 |
| # connected components | 98.253 | 61.461 | 50.530 | 47.680 |
| Size of largest component | 607.226 (80%) | 648.655 (85%) | 662.190 (87%) | 665.139 (87%) |
| # potential conflicts | 29 | 54 | 95 | 104 |

found with the lower settings. Table III shows how raising the distance of transitive includes effects the size of the call graph and the number of potential conflicts it identifies. In total, the call graph that supplied the numbers for Table III was computed from 50.424 files, yielding 760.196 named units. To identify potential conflicts, five different merges were compared, including 1, 7, 8, 53 and 66 changed C++ files respectively.

Increasing the transitive include distance beyond one did not result in a higher number of correctly identified conflicts, and is thus not worth the increase in false positives.

The distance between conflicting units describes the number of edges in the call graph, that separate the affected unit, in which the conflict occurs, from the conflicting units, which cause it. In the case of build conflicts, the affected unit always overlaps with one of the conflicting units (Figure 2), test conflicts do not share that restriction. Figure 7 shows the distances which were identified by our prototype. The 21 cases, in which the prototype was at least partially successful, contained 26 conflicts.

Six of the 26 conflicts were caused by changes to the same source code entity, resulting in distance of zero. In 15 cases, there was a single call between affected and conflicting units. The five remaining cases with distances between two and five upon closer inspection also had distance one in reality, but the prototype was unable to find them without increasing the allowed distance. This was the case because the prototype found longer, more complicated paths to the conflicting units when changed names of source code segments made the shorter paths unrecognizable in the call graph.

Using the parameters determined above, the performance of our prototype is shown in Table IV. A success requires, that all conflicting and affected units are correctly identified, and that the calls between them are accurate. In this case, the result could be used both for prediction and as assistance for developers to resolve the conflicts. A partial success occurs, when multiple conflicts are present, but not all were correctly

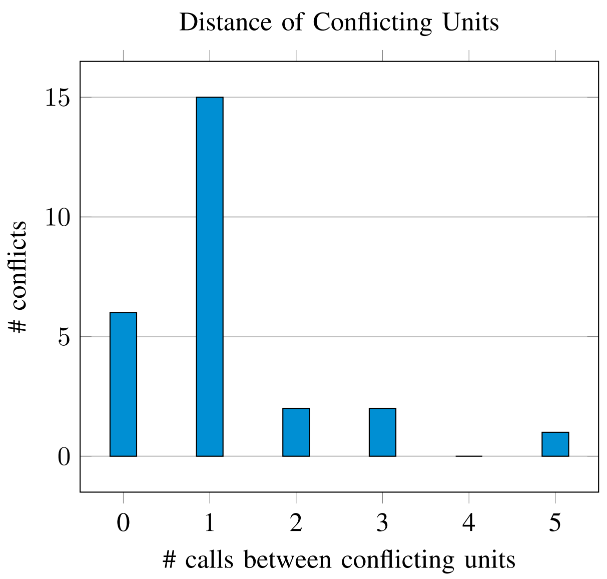


Fig. 7. Number of calls required to reach one conflicting unit from the other conflicting unit.

identified, or the sequence of calls is incorrect. The prototype still issues a correct warning, but the result is incomplete and is only partially useful for resolving the conflicts. Partial successes are rare, as they require multiple merge conflicts to be present within the same merged changeset. A failure requires that no conflict was identified correctly. In these cases, the merge conflicts were overlooked and no useful prediction could be made.

We split the result by the cause of the conflict and differentiate a signature change further by whether it effects the name of the unit. As our prototype identifies units by their name, changing or removing names negatively effects the performance more than changes to the return type or the arguments. Missing or incorrect include statements cannot be found by our prototype, as they occur outside of the units that are used as nodes in the call graph.

As partial successes also contribute correct predictions, we reach a recall rate of 62%. Test conflicts, which would be caused by changes to the internal logic, would fall into the same category as changes to the input and output, which shows significantly higher recall. Out of 79 executions performed to find test conflicts, we found 22 false positives (28%). This number is however inflated. Our prototype compares all merges in an execution pairwise and assumes, that all are being tested in parallel. To cover a large number of merges, we had to test merges performed over the course of one or two days in one execution. As a result, some of these merges could not have conflicted with each other. To get a better idea of the rate of false positives, we ran 22 executions with merges, that were currently under test. As this data was taken directly from the monitoring system, we know that these merges were tested in parallel. Of these 22 executions, 2 reported potential conflicts (9%). For comparison, the total test set (including both these

TABLE IV
PROTOTYPE SUCCESS-RATE FOR BUILD CONFLICTS BY CAUSE

| Cause of conflict | Success | Partial success | Failure |
|---|---|---|---|
| Name changed | 7 | 1 | 8 |
| Input or Output changed | 9 | 2 | 0 |
| Include | 0 | 0 | 4 |
| Duplicate definition | 2 | 0 | 1 |

22 runs and the previous, larger cases) contained on average 18.8 merges, the reduced test set of 22 runs averaged at 8.5 merges per execution.

## C. RQ3: How common are test conflicts?

Previous research on this topic is scarce and the available sources disagree by a large margin: Brun *et al.* [5] found test conflicts in 27% of the 399 merges they examined, which is 4.5 times as many as build conflicts. Kasi and Sarma [8] report 20% test conflicts and 9% build conflicts in 1158 merges, which also suggests that test conflicts are more common than build conflicts, though only by a factor two. By contrast, Accioly *et al.* [9] report only 5 test conflicts and 84 build conflicts out of 64,445 merges.

This leaves us with only a vague idea as to how many test conflicts we can expect. Given that 54 build conflicts were recorded in 22 months, and that Brun *et al.* found test conflicts to be 4.5 times more common, we could expect up to 11 test conflicts per month. Going by the results of Accioly *et al.* however, it is possible that there are no test conflicts to find, given that the amount of merges we can cover is limited.

In total, we scan for test conflicts in three phases:

Current samples
: 22 runs are performed on merges that are being tested while we started the analysis. This is a realistic scenario, that a tool like our prototype may be used to predict conflicts before they occur.

All successful merges from February 2019
: These merges were two months old at the time of analysis, which means that the developers may still remember some of the changes made during that month. This is relevant as we may not be able to decide, whether a test failure is truly the result of a merge conflict. In such a case, interviewing the developer could prove valuable. February is also far enough in the past, that most of the bug reports are resolved. This is helpful, as bug reports show the presence of a defect and the resolution can be used to determine the cause. Many defects are complex and cannot be quickly analysed by someone unfamiliar with the code. To cover a high number of potential conflicts, we rely on bug reports, rather than analysing the merged changes ourselves.

All successful merges from July 2018
: As these changes are over half a year old, it may be more difficult to receive information from the developers. July 2018 did however stand out as the month with the highest concentration of build conflicts. This

suggests that a lot of potentially conflicting merges were made, and could have lead to more test conflicts as well. It will also let us check whether the results of February are representative.

To verify whether a potential conflict reported by the prototype corresponds to a test conflict, we manually compare it to the bug reports. Depending on the number of potential conflicts, this can be a time-consuming process and a detailed investigation is not feasible. To limit the workload, we require that bug reports fulfil these conditions:

1) The bug report has been created after both conflicting branches have been merged into the master branch.
2) The bug report has been created no more than two weeks after the merge. Test conflicts should lead to test failures in the testing of other merges, or in post-submit test runs. Both kinds of testing occur frequently and regularly, therefore it should not take long for a bug report to be filed. Test conflicts not uncovered by this method are likely too subtle do find in a superficial search, as they may not be covered a test case.
3) The bug has been resolved by a fix. Determining whether a bug is the result of a test conflict or not will be very difficult without access to its fix. As such we choose merges far enough in the past, that most bug reports from that time have been resolved.

Among the bugs that satisfy these conditions, we perform several searches to find the ones relevant to us. First, we examine all bug reports that contain the word *conti*, which is short for continuous testing. This will cover test conflicts, that were not found due to the reduced test set used during the merging process. The number of such bugs is usually low enough, that all of them can be examined manually. We examine the comments and the files changed by the fix and look for file and unit names that appear as potential conflicts in the results of the prototype.

Next, we search for bug reports with comments containing file or unit names from potential conflicts. Here we cover conflicts that have been found while testing later changes. In particular, we are interested in failing tests, that seem to have no relation to the changes that are being tested. We once again use the fix and the comments to determine, whether potential conflict and bug report are related.

In total, we recorded 79 executions of our prototype, with 1489 merges. As merges within the same execution are examined pairwise, this results in 21829 comparisons. Using the previously determined settings for our prototype, namely a transitive include distance of one and direct calls from affected to conflicting units, 22 of 79 executions yielded potential conflicts. These potential conflicts were found in 362 of the 21829 pairs of merges.

After examining the potential conflicts according to the method described above, none of the potential conflicts were found to have been the cause of a test conflict. This suggests that test conflicts are significantly less common than build conflicts in context of SAP HANA. It is also possible that our findings are not indicative of the real frequency of test conflicts. This may be due to several factors:

Test conflicts exist, but were overlooked by our prototype
: It is possible that our prototype is ill suited for finding test conflicts. As we have no real test conflicts to evaluate its performance, we have done so under the assumption that test conflicts are similar to build conflicts.

Bug reports were overlooked
: Both potential conflicts and bug reports are too numerous to conduct a detailed analysis on a larger sample. A more detailed analysis or less restrictive selection of bug reports might have uncovered real test conflicts. This would require more time, leaving us with a much smaller sample size, possibly strengthening the next point.

Test conflicts did not occur in the selected timespan
: As mentioned above, 54 build conflicts were recorded in 22 months. Previous research suggests that test conflicts occur with a similar frequency. As such, a large timespan is essential to ensure that the number of test conflicts is representative of the overall frequency. We examined two months from separate phases of development, but if test conflicts are rarer than we expected, it is possible that we simply chose a sample that contained no test conflicts.

As we have mitigated these risks to the best of our ability, we conclude that test conflicts are significantly less common than build conflicts either in general or specifically in SAP HANA.

## V. Related Work

Related work in this field essentially covers two topics: empirical studies of higher order merge conflicts, and approaches to detect or prevent merge conflicts.

### A. Empirical studies

Brun, Holmes, Ernst and Notkin [5] examined nine open-source repositories containing 3,562 merges. They found that 16% of all merges contained textual conflicts. A subset of the repositories has been analyzed for higher-order merge conflicts, which yielded 6% build conflicts and 27% test conflicts. Kasi and Sarma [8] performed a similar study on four open-source repositories that found 42% of the merges resulted in conflicts, of which 13% were textual, 9% build and 20% test conflicts.

Leßenich, Siegmund, Apel, Kästner and Hunsen [10] solely focus on textual conflicts. In 21,488 merge scenarios they report a total amount of 11% conflicts, and 6% if considering only Java code. The difference between the total numbers and those considering only Java code suggest, that other parts of the repository such as build files or documentation significantly contribute to textual conflicts.

Accioly, Borba, Silva and Cavalcanti [9] perform a study into the feasibility of text-based conflict predictors. They identify higher-order merge conflicts to select appropriate projects for closer inspection. In 422 projects containing 64,445 merges, they find 84 (0.13%) build and 5 (0.008%) test conflicts. They also comment on the significantly higher numbers reported in [5], [8]: In contrast to the previous two, this study disqualifies all conflicting merge scenarios, if the state prior to the merge already contained errors to avoid counting the same conflicts multiple times. Besides this, [5], [8] perform the merges locally which may have caused problems, such as unresolved dependencies, which Accioly, Borba, Silva and Cavalcanti filter out. By their own admission, the reported numbers are likely too low, but can serve as a lower bound.

Accioly, Borba and Cavalcanti [11] also find 5.91% textual conflicts in Java code of 70,047 merge scenarios. These conflicts are identified using a semi-structured merge tool, which resolves numerous conflicts line-based tools cannot, such as spacing issues. The relatively low number of textual conflicts can be explained by this more powerful merge tool.

In summary, previous studies have found textual conflicts in 11% - 16% of merge scenarios. Later studies also report textual conflicts only in the source code at around 6%.

Higher-order merge conflicts are harder to identify. This often involves building and testing the source code locally which is both time-consuming and error-prone, as external factors can contribute to errors not present in the source code [10]. The reported numbers range between 29% and 33%, but valid concerns are raised about the process of these studies [10], suggesting the real number is far lower. Our own findings show build conflicts in 0.41% of 14270 merges, and no test conflicts in 1489 merges. This is comparable to 0.13% build and 0.008% test conflicts reported by Accioly Accioly, Borba, Silva and Cavalcanti [9], but in stark contrast to Brun, Holmes, Ernst and Notkin [5] (6% build and 27% test conflicts) and Kasi and Sarma [8] (9% build and 20% test conflicts). Where build and test conflicts were differentiated, [5], [8] report test conflicts as more common, while [10] suggests more build conflicts. These differences are likely due to the external factors mentioned above.

## *B. Conflict detection and prevention*

Palantír [12] by Sarma, Noroozi and van der Hoek is an early tool aiming to avoid merge conflicts by increasing awareness. This is done by recording which artefacts of a repository each developer has interacted with, and notifying other developers making changes to the same artefact.

Dewan and Hegde developed CollabVS [13], which works similar to Palantír, but records not only which files, but also which classes or methods a developer is working on. This is achieved by recording their cursor position and activity using the IDE. It also considers dependencies such as classes depending on super classes. A different paper [14] with more focus on the implementation specifics reveals that dependencies are found using binary analysis.

Hattori and Lanza developed the tool Syde [15]. It creates an Abstract Syntax Tree (AST) of a program for each developer to determine conflicts by comparing changed AST nodes.

Brun, Holmes, Ernst and Notkin also pursue awareness with their tool Crystal [1]. Crystal makes use of speculative analysis [16] to detect merge conflicts. It does so by continuously merging, building and testing the current code of a developer against the master repository. Both textual and higher-order conflicts can be found reliably, because they actually occur in the background.

WeCode by Guimarães and Silva [2] also performs continuous merges. They use structural merging to avoid sending notifications about easily resolved textual merges. Test conflicts are identified by two different methods: Conflicts covered by automated test cases are found by running the tests. If no appropriate test case is available, an abstract semantic graph is used to identify dependencies and highlight potential conflicts.

Kasi and Sarma take a different route with their tool Cassandra [8]: Instead of detecting conflicts as they occur, developer tasks are scheduled in a way that conflict-prone activities are not carried out concurrently. A dedicated IDE plug-in determines which source code artefacts are assigned to each task, and a separate dependency analysis tool finds associated files.

Semex by Nguyen, Nguyen, Dang, Kästner and Nguyen [17] combines different code versions into a single program with variable aware execution. Where the code diverges, the common program has a branching path for each version, so that each distinct path from beginning to end is one possible combination of changes. Tests are then run on every combination of paths, which identifies test conflicts without pairwise merging and testing all existing branches. It also identifies which set of changes causes the conflicts..

Dias, Polito, Cassou and Ducasse [18] suggest using change impact analysis to find dependencies between code artefacts in their technique DeltaImpactFinder. The approach compares the impact of a patch applied to multiple version of a repository. Should the same patch introduce a different set of dependencies to the different versions, semantic merge conflicts are likely to occur.

Ahmed, Brindescu, Mannan, Jensen and Sarma [19] examine the relationship between merge conflicts and code smells. Methods showing particular code smells are more likely to be involved in merge conflicts.

Pastore, Mariani and Micucci [3] present Behavior Driven Conflict Identification (BDCI) which focuses on a program's behavior. BDCI identifies all methods, that have been modified and monitors their entry and exit points during test executions to derive possible input and output values. In the case of conflicting changes to the observed values, higher-order conflicts are likely the cause.

Accioly, Borba and Cavalcanti [9] examine the potency of static conflict predictors. They make use of a semi-structured merge tool and their own ConflictAnalyser from a previous work [11].

Leßenich, Siegmund, Apel, Kästner and Hunsen [10] examine the effectiveness of seven conflict indicators suggested by developers such as the activity of branches and the number of changed files.

Sousa, Dillig and Lahiri [20] formally define semantic conflict freedom and build a tool to verify whether a merge satisfies these conditions. This tool merges the base version of a file, two changesets and a merge candidate to identify differences. The common parts of all four source code versions are then considered a single program with holes, that at least one of the versions fills differently from the rest. Using relational postconditions, their tool finds differences in the behavior of the four versions.

Finally, Xing and Maruyama [21] propose an approach to automatically repair behavioral merge conflicts once they are found. They automate the search for a merged version of two branches, which satisfies test cases from both branches using an automated program repair technique.

Many of the older tools presented here record which parts of the source code developers make changes to and look for collisions between team members in real-time [8], [12], [13], [15]. Changes to the same or dependent files can also be identified using the committed changesets which proved to be a solid basis for analysis [9].

An evolution of this approach is shown in [1]–[3]: by compiling and testing the potential merges ahead of time, the rate of false positives can be drastically reduced. This comes at a significant cost regarding time and processing power, as build and test scripts have to be run frequently. Some efforts have been made to mitigate this cost, such as in [17], [20]. While these approaches present a significant improvement for small and medium projects, we consider a scale at which the build process alone would be prohibitively time-consuming.

Static analysis avoids the problem of large build and test scripts. While the indicators examined in [10] proved ineffective, [9], [18], [19] suggest methods that may be applicable. We particularly built on the predictors suggested by Accioly, Borba, Silva and Cavalcanti [9].

## VI. Conclusion

We propose and implement an approach to predict higher-order merge conflicts using static analysis and evaluate it on the SAP HANA project. The use of static analysis along with the reusing of previously parsed data allows our prototype scaling to large projects, where building and testing the program is too time-consuming to keep up with the rate of development. The trade-off is a high computation time during the first execution, or in case the previously parsed data is lost. Our prototype makes use of the C++ parser srcML to create a static call graph and identifies potential conflicts on the level of units, which can be uniquely identified by their name. The distance between conflict parties can be fine-tuned both on unit and file level, and we empirically determine the most promising settings for our repository.

This prototype is evaluated on known build conflicts and reaches a recall rate of 62%, with the success being largely dependent on whether or not the names of the conflicting units have changed. On a small selection of 22 real-life executions with 8.5 merges on average, the rate of false alarms was 9%. For test conflicts (which are likely to be caused by changes to the behaviour rather than the name of a code segment) our prototype should be able to achieve a higher recall rate.

Despite these promising results, we found no test conflicts in over two months of development work. In the same timespan, eight build conflicts were recorded. These results suggest that in our setting test conflicts are much rarer than build conflicts.

In conclusion, our prototype shows promising results on build conflicts while mostly avoiding the anticipated problem of false positives despite its numerous limitations. However, for small and medium sized projects with good test coverage dynamic tools might provide significantly more accurate results at comparable or better runtime.

To validate our findings (or lack thereof) on test conflicts, one could apply this approach to find test conflicts reported by Brun, Holmes, Ernst and Notkin [5] and Kasi and Sarma [8]. This would help to determine whether the lack of test conflicts was due to specific properties of SAP HANA, or whether it can be traced back to our approach.

The prototype itself also could be improved in various ways. One option is to use a more reliable call graph. By creating a call graph during the build process one would loose the speed of static analysis, but it would still be possible to skip the testing process, which may be enough depending on the size of the project. The call graph could also be refined by considering the type of unit during graph construction (our current prototype uses only names to identify units).

## References

[1] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Crystal: Precise and unobtrusive conflict warnings," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 444–447, ACM, 2011.

[2] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, (Piscataway, NJ, USA), pp. 342–352, IEEE Press, 2012.

[3] F. Pastore, L. Mariani, and D. Micucci, "Bdci: Behavioral driven conflict identification," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), pp. 570–581, ACM, 2017.

[4] T. Wuensche, A. Andrzejak, and S. Schwedes, "Prototype sourcecode." https://github.com/tbwuensche/Detecting-Higher-Order-Merge-Conflicts-in-Large-Software-Projects.

[5] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 168–178, ACM, 2011.

[6] M. L. Collard, J. I. Maletic, M. Decker, B. Bartman, C. Newman, D. Guarnera, P. P. Leyden, C. Bryant, V. Zyrianov, M. Weyandt, H. Guarnera, B. Kovacs, P. E. Jordan, A. Myers, T. Sage, and K. Swartz, "srcml." https://www.srcml.org/. Accessed: 18.12.2018.

[7] M. Mahmoudi, S. Nadi, and N. Tsantalis, "Are refactorings to blame? an empirical study of refactorings in merge conflicts," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019* (X. Wang, D. Lo, and E. Shihab, eds.), pp. 151–162, IEEE, 2019.

[8] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 732–741, IEEE Press, 2013.

[9] P. Accioly, P. Borba, L. Silva, and G. Cavalcanti, "Analyzing conflict predictors in open-source java projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, (New York, NY, USA), pp. 576–586, ACM, 2018.

[10] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen, "Indicators for merge conflicts in the wild: Survey and empirical study," *Automated Software Engg.*, vol. 25, pp. 279–313, June 2018.

[11] P. Accioly, P. Borba, and G. Cavalcanti, "Understanding semi-structured merge conflict characteristics in open-source java projects," *Empirical Software Engineering*, vol. 23, p. 2051, Aug. 2018.

[12] A. Sarma, Z. Noroozi, and A. van der Hoek, "PalantÍr: Raising awareness among configuration management workspaces," in *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, (Washington, DC, USA), pp. 444–454, IEEE Computer Society, 2003.

[13] P. Dewan and R. Hegde, "Semi-synchronous conflict detection and resolution in asynchronous software development," *ECSCW 2007*, Jan. 2007.

[14] R. Hegde and P. Dewan, "Connecting programming environments to support ad-hoc collaboration," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Software Engineering*, pp. 178–187, Sept. 2008.

[15] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, (New York, NY, USA), pp. 235–238, ACM, 2010.

[16] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis: Exploring future development states of software," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, (New York, NY, USA), pp. 59–64, ACM, 2010.

[17] H. V. Nguyen, M. H. Nguyen, S. C. Dang, C. Kästner, and T. N. Nguyen, "Detecting semantic merge conflicts with variability-aware execution," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), pp. 926–929, ACM, 2015.

[18] M. Dias, G. Polito, D. Cassou, and S. Ducasse, "Deltaimpactfinder: Assessing semantic merge conflicts with dependency analysis," in *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '15, (New York, NY, USA), pp. 8:1–8:6, ACM, 2015.

[19] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma, "An empirical examination of the relationship between code smells and merge conflicts," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '17, (Piscataway, NJ, USA), pp. 58–67, IEEE Press, 2017.

[20] M. Sousa, I. Dillig, and S. K. Lahiri, "Verified three-way program merge," *Proc. ACM Program. Lang.*, vol. 2, pp. 165:1–165:29, Oct. 2018.

[21] X. Xing and K. Maruyama, "Automatic software merging using automated program repair," in *Proc. IEEE 1st Int. Workshop Intelligent Bug Fixing (IBF)*, pp. 11–16, Feb. 2019.