Symbolic Execution to Detect Semantic Merge Conflicts

Anonymous

***Abstract*—Collaborative software development depends on managing multiple versions of a program which requires mechanisms to merge program versions to eventually deploy a single executable. Merging program versions can be challenging as conflicts can arise. The most challenging form is a semantic conflict, which introduces unintended behaviour in the resulting executable while merging.**

In this paper, we develop an approach that detects such semantic merge conflicts by symbolic execution. We define the program semantics as path conditions, produced by a symbolic executor, and check whether the conditions satisfy established rules that reflect a merge conflict. Our usage of symbolic execution to check these rules is novel. We evaluate the correctness of our approach through mutation testing, and evaluate it empirically by applying the approach to real-world merges sampled from GitHub. We also discuss what challenges arise in the empirical evaluation, including the problems i) that semantic merge conflicts are rare in the wild, ii) and, even in retrospection, hard to find using standard search mechanisms. Our evaluation shows that in specific cases, our approach using symbolic execution is a promising extension to existing mechanisms to merge conflict detection.

***Index Terms*—three-way merge, semantic merge conflict, symbolic execution, program behaviour**

I. Introduction

When a team of software developers collaboratively works on a program, there typically exist several program versions in parallel [3], [30], [31]. This enables one developer to work on a version with a new feature, and another developer to implement a fix [3]. The versions are stored in version control systems (VCS) [11]. However, the team eventually needs to merge the program versions to deploy a single executable. Merging is a challenge since conflicts between program versions can occur [4], [33].

The most challenging form is a semantic conflict, which introduces unintended behaviour in the resulting executable while merging (see [22], which distinguishes textual, syntactic, and semantic conflicts).

A. Semantic Merge Conflict

We introduce an example of a semantic merge conflict that cannot be detected on a textual or syntactic level. It can be detected by our approach using symbolic execution.

```
1 public int myAdd(int x, int y) {
2   int z = x + y; // Updated to ‘x + y + 1’ in version A
3   return z; // Updated to ‘z + 1’ in version B
4 }
```

The example shows an original program version where the `myAdd` method returns the sum of `x` and `y`. One developer notices that there is an ‘off-by-one’ error and changes line 2 to `int z = x + y + 1` in program version A. Another developer resolves the same problem, but changes line 3 to `return z + 1` in program version B.

The merged version M includes the changes done in program version A and B and returns the sum of `x` and `y` plus two. The behaviour of the program has changed in an unintended way, resolving an ‘off-by-two’, but not an ‘off-by-one’ error. This happens despite the semantic change to version A and B being the same and despite both program versions behaving the same. **Such a semantic merge conflict cannot be detected by Git’s merge or by a compiler.**

B. Detecting Semantic Merge Conflicts by Symbolic Execution

In this paper, we present an approach to detecting semantic merge conflicts by symbolic execution (see [8], [21] for symbolic execution). Our approach correctly classifies the previous example as a semantic merge conflict.

To this end, our approach defines the program semantics as path conditions, produced by a symbolic executor, and checks whether the conditions satisfy established rules that reflect a merge conflict. Our usage of symbolic execution to check these rules is novel.

We develop an automated prototype that helps developers to avoid semantic merge conflicts.

C. Evaluation

Semantic merge conflicts are rare in the wild, which complicates the evaluation. To evaluate our approach, we investigate the following two research questions:

- **RQ1**: Can we classify semantic merge conflicts retroactively by heuristics computed on the full revision history?
- **RQ2**: Can we classify semantic merge conflicts proactively by symbolic execution?

We use the retroactive classification (RQ1) to get a pool of candidates with a realistic chance of being a semantic merge conflict. The pool is used for the evaluation of our proactive approach by symbolic execution (RQ2). Using a random sample from GitHub is unrealistic due to the vanishingly small number of true semantic merge conflicts on GitHub.

Our evaluation shows that in specific cases, our approach using symbolic execution is a promising extension to existing mechanisms to merge conflict detection.

D. Contributions

- We present an approach to detect semantic merge conflicts by symbolic execution.
- We implement a prototype for detecting Java merge conflicts that is used for the evaluation of our approach on synthetic and empirical data.
- We present an alternative retroactive approach to detect semantic merge conflict on the full revision history.

E. Roadmap

Section II starts with the necessary background for introducing our approach; Section III discusses the related work; Section IV explains our approach that uses symbolic execution to detect semantic merge conflicts; Section V describes a prototype implementation of our approach, listing technical details and limitations; Section VI evaluates the prototype; Section VII concludes the paper.

II. Background

A semantic merge conflict is a conflict in the program behaviour that happens during merging. In this paper, we focus on an abstraction of the program behaviour produced by symbolic execution (see [8], [21]).

We start with an introduction to symbolic execution and merge conflicts. The background is required for the description of our approach in Section IV.

A. Symbolic Execution

Symbolic execution is a static analysis technique in which the input values of a program are replaced by symbolic variables. While executing the program, different constraints are placed on the symbolic variables (e.g., by branching conditions or variable assignments). The constraints relate the program input and output values and are gathered in a *path condition*.

Path conditions are abstractions of the program behaviour. To give an example, we consider the `myAdd` function from Sec. I again. In its original program version (without any changes) there is one path condition, i.e., $z = x + y$ where variable z is the program output and variables x and y are the program inputs.

Path conditions can be used in subsequent analysis steps by constraint solvers, making them relevant to our approach. In our case, such an analysis step classifies a conflict in a merge using the path conditions.

B. Version Control Systems

Version control systems (VCS) help to develop and maintain programs as a team by managing different program versions and keeping track of the evolution [11]. VCS such as Git, SVN, and Mercurial, describe the evolution of the software as a directed acyclic graph (DAG). Nodes represent snapshots of the software. They are also called versions, commits, or revisions. Edges reflect actions by developers, such as, committing, branching, or merging.

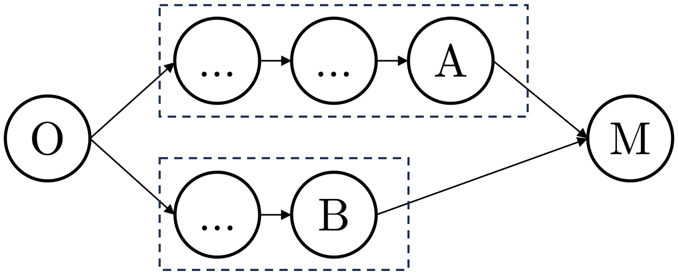

Fig. 1. The four parts involved in a merge. Branches A and B share a common ancestor commit O. The changes are combined into the merge commit M.

C. Merging

Diverging program versions lead to branches in the directed acyclic graph. Branching may be used for many reasons, for instance, to separate the development of certain program features [3]. Such separation is crucial to develop and maintain the program as a team.

However, branching introduces the problem of *merging*. Here, different program versions need to be combined again into one single version. Fig. 1 shows such a situation, where two branches with the same origin are merged into one single program version.

In particular, the relevant nodes in the acyclic graph, reflecting program versions, are:

- **The Origin (O)**: The origin is the program version that both branches base their work on.
- **Branch A and B**: The two branches evolve from O by individual efforts. They can consist of one or more program versions. The last version in each branch is the one relevant to our approach. We label them as A and B.
- **The Merge (M)**: Both branches are merged into a single program version M.

The scenario depicted in Fig. 1 is called a *three-way merge* because it considers the origin and the (last version of the) two branches. It was introduced with the Unix `diff3` program in the late 1970s. It is the current standard merge strategy for Git. Combining more than two branches at once (*n-way merge*) is uncommon, so we exclude it from the following discussion.

D. Merge Resolution and Merge Conflicts

Merging the different versions of a program is not trivial and cannot always be automated [4], [6], [24].

In this background section, we will explain the typical process of merging in Git. We will discuss different forms of conflicts that may arise before a merge can be accepted. We depict this process in Fig. 2. The diagram already contains our prototype that we will discuss in Sec. IV.

1) Textual Merge (Git's Default): Merging is often done using a textual representation of the program (typically the source code). We focus on this textual merge strategy because it is the default for Git. Git uses the `diff3` algorithm. Other textual approaches are discussed in [5], [25], [38].

An automatic textual merge may not always be possible. For example, if both branches changed the same part of the

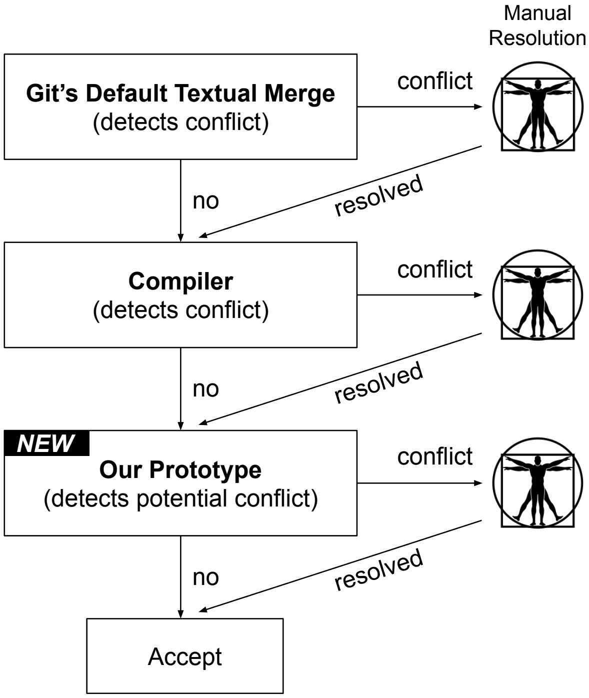

Fig. 2. The semi-automated process to check the conflict freedom of merge commits. The process eventually leads to an accepted merge.

text in different ways, there is no way of knowing which one to prioritise. This requires a manual resolution of a *textual merge conflict*. Git, for instance, detects and forces the user to resolve the conflict.

In this paper, we use the terms *detecting* and *classifying* a conflict interchangeably.

2) Compiler: When the textual conflict is resolved, a syntactic conflict may remain. The compiler typically catches such conflicts. For example, a conflict with an extra opening (or closing) brace in the code can be detected by a compiler, but not by a textual method.

While the compiler provides an accurate classification of syntactic conflicts, the resolution is typically done manually when working with Git's default merge.

There are also syntactic merge approaches (typically working on ASTs) that may automatically resolve (or prevent) syntactic conflicts (see [1], [7], [41]).

3) Semantic Conflicts: What remains are the semantic conflicts, where a classification of a conflict depends on the behaviour of the program. Semantic conflicts are challenging. **In this paper, we focus on how to detect semantic merge conflicts using symbolic execution.**

III. Related Work

We continue with an in-depth discussion of the related approaches. We position this paper as the first that classifies (or detects) semantic merge conflict by symbolic execution.

We introduce the following three dimensions to structure the related work discussion. The first two reflect the structure of this section.

Program analysis vs. data-driven: We see differences in how approaches classify or resolve conflicts. Approaches can be based on program analysis. This is the case for our approach. Other approaches are 'empirically' motivated using collected data or existing knowledge on merges.

Classification vs. Resolution: This section distinguishes between approaches that: i) classify merge conflicts for a subsequent manual resolution (i.e., detection), and ii) approaches that automatically resolve (or prevent) conflicts. Approaches for classification and resolution apply related techniques. Our approach provides a classification.

Semantic vs. non-semantic: Aspects of 'merge conflict types' have already been discussed in Sec. II-D.

We will focus on semantic merge conflicts in this section, but we will mention alternatives, if they are part of relevant related work.

A. Program Analysis Approaches

We start with a discussion of approaches that do a classification of conflicts based on a program analysis. Our approach falls into this category.

1) Classification of Conflicts: The authors of [35] define semantic conflicts as the introduction of unwanted behaviour. This is comparable to our definition. They introduce SafeMerge, a tool that considers a merge as four modifications to a common base. They apply a lightweight analysis to the shared parts, and verify modifications (in particular the return values of a program). SafeMerge is evaluated on real-world code, but the authors note that a semantic conflict is hard to define objectively. They evaluate against their own definition of a semantic merge conflict, inherent to their approach (Definition 4.4, page 7, in [35]). Our qualitative evaluation attempts to mitigate this by judging a conflict through the presence of bug fixes for a merge. Our definition of a conflict is closely related to the definition used by SafeMerge (see Sec. IV-B). However, we use symbolic execution to instantiate the definition.

Da Silva et al. [10] consider static analysis tools (like symbolic execution) as too heavyweight. Instead, they use test generation. Tests are generated using the program versions A and B to capture the behaviour. The tests are then executed on versions O and M. When a test fails in O and M, but succeeds in A (or B), then there is a possible conflict. Da Silva et al. report on many false negatives produced by their approach. This reflects the complexity of classifying semantic merge conflicts.

Wuensche et al. [42] classify build and test conflicts. Their approach creates a directed call graph of the code and its modifications to detect a conflict if: i) changes are made on the same call graph node, ii) there is a path from one change to another, or iii) there is a path from an unchanged node to two changed nodes. This is clearly different to symbolic execution and more on the syntactic level. In an evaluation, Wuensche et al. show that the approach helps to catch build conflicts.

However, test conflicts are rare in the sample that Wuensche et al. use for the evaluation. Hence, statements on whether the approach detects test conflicts are not possible. This reflects a challenge of our evaluation that shows that semantic conflicts are rare. We resolve this by producing a pool of promising semantic conflict candidates by a retroactive search method which operates on the full revision history. We use this sample of candidates in the evaluation of our approach.

Pastore et al. [29] use specification mining to generate behavioural program models for versions O, A, and B. They run a program's tests to collect values for variables, after which pre- and post-conditions are derived from the observed values. If changed conditions differ between A and B, according to a constraint solver, a conflict is reported. Our approach does not require an existing test suite and also takes into account the behaviour of version M.

2) Resolution of Conflicts: Another branch of related work focuses on automatically resolving conflicts.

MrgBldBrkFixer [37] is a tool that aims to automatically resolve conflicts, such as inconsistent symbol renaming. When identifying a renamed symbol that causes the build to break, it filters the commits after the last correct build and uses them to generate patches to resolve the inconsistency.

Horwitz et al. [19] use program dependency graphs and program slicing to create graphs representing the changed code in both branches. If the graphs do not overlap, there is no conflict. Merge conflicts are automatically resolved by merging overlapping graphs, deriving the merged code from the merged graphs. This is a more syntactic resolution of a conflict.

B. Data-driven Approaches

Other approaches are 'empirically' motivated, using collected data or existing knowledge on merges, to classify or resolve conflicts.

1) Classification of Conflicts: The Bucond tool, presented in [39], creates a program entity graph for version O, A, and B. The program entity graph contains standard AST information but also def-use relations. A comparison of the different graphs is searched for predefined conflict patterns. Conflict patterns reflect very specific information that we consider as empirically motivated. A merge conflict is reported if a pattern matches. Compared to our approach using symbolic execution, Bucond is limited to predefined patterns.

Somewhat related is the classification of pull requests. Models that predict the acceptance of pull requests are trained on empirical data. An example can be found in [40]. However, non-acceptance of pull requests may not necessarily be caused by a merge conflict.

2) Resolution of Conflicts: DeepMerge [14] uses a machine learning model to suggest resolutions for a textual merge conflict. MergeBERT [38] improves on this approach, also targeting textual merge conflicts. Learning from existing data on resolutions is also employed by, for example, Pan et al. [27] and by Almost Rerere [18].

Our approach does not rely on data. However, we assume the combination of ML with symbolic execution to be promising future work.

IV. Detecting Merge Conflicts by Symbolic Execution

This section describes our approach to detecting merge conflicts by symbolic execution. We structure this section as follows:

- Sec. IV-A describes a merge conflict in terms of a property P. The property is a placeholder. We later define the property such that it reflects the semantics of a program version.
- Sec. IV-B formulates basic rules that describe the merge conflict given a property P. This leads to a classification as merge conflict. The rules are inspired by the related work.
- Sec. IV-C instantiates the property P using symbolic execution. This results in a classification as a merge conflict based on the semantic notion of symbolic execution.
- Sec. IV-D describes how to decompose a merge conflict that arises from such a definition, into a more fine-grained notion of a conflict.

A. Property P

Our definition of a conflict is based on a property that we denote as P. The property P is a placeholder. It is specific to the four program versions involved in a merge. This can be described as a 4-tuple: (P_O, P_A, P_B, P_M).

For illustration, we consider a concrete example in which we analyse the presence of a particular line of code. We define P as a boolean function, checking whether the line is present in the origin O, the branches A and B, and the merge M. An example is a program where branch A adds a line which is not present in the origin O or in branch B. If the line also shows up in the merge, then the tuple is $(\mathit{False}, \mathit{True}, \mathit{False}, \mathit{True})$.

In Sec. IV-C, we switch to a semantic notion of such a property, created by symbolic execution.

B. Merge Conflict of Property P

We now define a merge conflict as a function of the 4-tuple for property P. We also report on the intuition why such a violation might be a conflict. In Sec. VI, we evaluate this definition.

A merge conflict arises if one of the following three constraints is violated:

- **Conflict Freedom A (CF-A)**: We face a merge conflict if $(P_O \neq P_A) \to (P_A = P_M)$ is violated. The intuition is that if the property differs between the common origin O and the branch A, i.e., $P_O \neq P_A$, then we expect the merge property P_M to reflect this, by being equal to property P_A. This is $P_A = P_M$.
- **Conflict Freedom B (CF-B)**: Analogous to CF-A.
- **Conflict Freedom A and B (CF-AB)**: We face a merge conflict if $(P_A = P_B) \to (P_A = P_B = P_M)$ is violated. The intuition is that if the property is the

same in both branches, i.e., $P_A = P_B$, then we also expect this property to be present in the merge M, i.e., $P_A = P_B = P_M$.

The definition of equality ($=$) and inequality ($\neq$) depends on the way we define the property P. The arrow $\rightarrow$ denotes a logical implication.

In our concrete example of a line of code, where the property is defined as a boolean function, the equality ($=$) is the standard boolean operator.

One reason for a violation of CF-A is that a line is added in branch A, but the line is not present in the merge M. One reason for a violation of CF-AB is that a line is added by both branches, but it is not present in the merge M.

Version control systems, like Git, already apply comparable rules when performing a merge requested by the developer. A merge is constructed in a manner that the above rules are not violated.

In the following, we switch from our basic example to a semantic property of the program and thereby to a semantic conflict.

C. Semantic Merge Conflict

We start with a simplified discussion of a program with a single method that is changed independently in two branches. Both versions are eventually merged. In Sec. IV-D, we explain how to extend this idea to make statements about different structural elements of the program and thereby decompose a semantic conflict.

Symbolic execution typically results in multiple path conditions, due to the controls structures in a program. We avoid this detail in this section and consider a program with just one path condition. We then extend it to multiple paths in Sec. IV-D.

Our approach uses a symbolic execution engine to define the property P as a path condition. A constraint solver is used to define equality between path conditions. The property P thereby captures the behaviour of the method.

Equality and inequality, previously denoted as $=$ and $\neq$, are now defined using a constraint solver. To emphasise the difference, we denote the equality proven by the constraint solver as $\Leftrightarrow$ and $\not\Leftrightarrow$. The symbol $\rightarrow$ still denotes a logical implication.

Our rules thereby instantiate as follows:

- **CF-A**: $(P_O \not\Leftrightarrow P_A) \rightarrow (P_A \Leftrightarrow P_M)$
- **CF-B**: Analogous to CF-A.
- **CF-AB**: $(P_A \Leftrightarrow P_B) \rightarrow (P_A \Leftrightarrow P_B \Leftrightarrow P_M)$

In essence, the rules state that if we face a semantic change in one of both branches, we expect the new behaviour to be present in the merge too (CF-A or CF-B). If both branches are semantically equivalent, we expect them to be semantically equivalent to the merge (CF-AB).

If we go back to our semantic example from the introduction in Sec. I, we have the following path conditions as properties for O, A, B, and M.

$$\begin{aligned} P_O &: r = x + y \\ P_A &: r = x + y + 1 \\ P_B &: r = x + y + 1 \\ P_M &: r = x + y + 2 \end{aligned}$$

Substitution of the properties into the rules produces a semantic merge conflict because all rules are violated.

D. Decompose the Conflict

When applying the previous approach to the program as a whole — this can be imagined as applying it to the single main method — we will almost always detect a conflict. Such a semantic conflict lies in the nature of branching, since both branches are intended to make semantic changes to the program.

Based on the idea of locality of behaviour, we try to solve this by decomposing the program into its parts. This enables us to get a more meaningful insight into a semantic conflict, by reporting on the number and locations of violations for the parts.

To this end, we change our rules to operate on 4-tuples of sets. The set needs to respect the definition of the equality that we plug into our approach. The rules are adjusted as follows:

- **CF-A**: If $P_A \setminus P_O \subseteq P_M$ is violated, we face a conflict. All elements that are added by A, that are not previously present in O, need to be contained in M.
- **CF-B**: Analogous to CF-A.
- **CF-AB**: If $P_A \cap P_B \subseteq P_M$ is violated, we face a conflict. All elements that are contained in both A and B need to be part of M.

For a single element set, the rules correspond to the rules we defined in Sec. IV-B. This set notation can be transferred into statements on which elements are missing in the merge, giving the locations, and quantifying violations.

For completeness, we list the set operations with the adjusted equality defined by a constraint solver.

$$A \setminus B := \{a \in A \mid \nexists b \in B : b \Leftrightarrow a\} \quad \text{[Set Difference]}$$

$$A \cap B := \{a \in A \mid \exists b \in B : b \Leftrightarrow a\} \quad \text{[Set Intersection]}$$

$$A \subseteq B \textit{ iff } \forall a \in A \; \exists b \in B : a \Leftrightarrow b \quad \text{[Subset]}$$

For our semantic approach to detect conflicts, we partition the behaviour of the program. This is done by a partitioning of the input space of the program that is specific to symbolic execution.

Consider the example in Listings 1 to 4 with control structures and more than one path condition given by a symbolic executor. The code includes two checks for a divide-by-zero error. Branch A and B do not agree on which check to remove to avoid this redundancy. Hence, the result is missing a check for the divide-by-zero error after merging, i.e., a semantic conflict.

```
1 int div(int x, int y) {
2   if (y == 0) // Removed in A
3     return 0; // Removed in A
4
5   if (y != 0) // Removed in B
6     return x / y;
7   else         // Removed in B
8     return 0; // Removed in B
9 }
```

Listing 1. The original **version O** has a redundant safety checks (in particular divide-by-zero). Branches A and B remove one check. However, they do not agree on which check is removed. In M, both checks are missing, so the safety check semantics is missing.

```
1 int div(int x, int y) {
2   if (y != 0)
3     return x / y;
4   else
5     return 0;
6 }
```

Listing 2. **Branch A** removes lines 2–4 from O.

```
1 int div(int x, int y) {
2   if (y == 0)
3     return 0;
4
5   return x / y;
6 }
```

Listing 3. **Branch B** removes lines 5, 7, and 8 from O.

```
1 int div(int x, int y) {
2   return x / y;
3 }
```

Listing 4. In **version M**, both checks are missing, so the safety check semantics are missing.

Symbolic execution handles the control structures by adding new path conditions (e.g., one for the 'then' and another of the 'else' of an 'if'). Thus, it partitions the input space of the method. The 4-tuple of sets of path conditions for the program versions look as follows for this example:

$$\begin{aligned}
P_O &= \{y = 0 \land r = 0,\ y \neq 0 \land r = x/y\}\\
P_A &= \{y = 0 \land r = 0,\ y \neq 0 \land r = x/y\}\\
P_B &= \{y = 0 \land r = 0,\ y \neq 0 \land r = x/y\}\\
P_M &= \{r = x/y\}
\end{aligned}$$

Applying the conflict freedom rules to these sets results in the following: For CF-A $P_A \setminus P_O = \emptyset \subseteq P_M$, so CF-A is not violated (analogous for CF-B). For CF-AB, however, $P_A = P_B = (P_A \cap P_B) \not\subseteq P_M$, so CF-AB is violated. The path condition $y = 0 \land r = 0$ reflects that the check for a divide-by-zero error is missing. We report a semantic conflict.

In the following section, we will discuss the technical details of our prototype implementation of this approach to detecting semantic conflicts. In Sec. VI, we will evaluate our definition of a semantic conflict on synthetic and real-world examples. Some examples will be discussed in depth.

V. Technical Details

We implement a prototype of the approach that we need for the evaluation in Sec. VI. This section describes the technical details of the prototype and its limitations. We separate such details from the description of our approach, since they can be ignored on a conceptual level. However, they impose technical limitations on the following-up evaluation that is based on the prototype.

Symbolic execution is technically challenging and strongly dependent on existing technology. We reuse a combination of Symbolic PathFinder [28], Gumtree [16], and Z3 [12]. We adapt Symbolic PathFinder and Gumtree as described in this section.

A. Aligning Variables in Path Conditions

The path conditions for each program version result from different runs of a symbolic executor. However, there is no clear relationship between symbolic variables created in the different runs. To show an equivalency between the path conditions, we create a mapping between the symbolic variables for path conditions for different program versions.

Consider the example of two path conditions, $x_O < 10$ and $x_A < 10$, resulting from an analysis of O and A. Variables x_O and x_A are symbolic and created during analysis. Without further information on the equivalence of variables, one cannot show the equivalence $x_O < 10 \Leftrightarrow x_A < 10$. An extra conjunct $x_O = x_A$ is needed.

We use a syntactical analysis as a preprocessing step in our approach that resolves this problem. The step finds matches between AST nodes across different program versions. When using the constraint solver to find an equivalence, our prototype uses this mapping to add equalities between symbolic variables like $x_O = x_A$.

Our prototype uses Gumtree [16] for this step. Gumtree finds mappings between nodes of the ASTs. Our prototype runs Gumtree on every combination of the ASTs of program version O, A, B, and M.

B. Symbolic Execution Engine

For the symbolic execution, we use Symbolic PathFinder (SPF) [28], an extension to the Java PathFinder (JPF) [17]. SPF is still used in recent research, for example in HyDiff [26].

SPF does have some shortcomings. At the time we started using SPF, it supported Java 8. New language features have been added to Java since then. SPF also does not support all Java constructs. It does not model the entire standard library and misses some language features, such as `try-catch`.

1) Symbolic Variable for Output: To detect equivalent path conditions (previously denoted as $\Leftrightarrow$), we are interested in constraints on the symbolic input and output variables of a program. SPF does not create constraints on the output, however. We modified SPF to create an extra symbolic variable for the output returned by a method.

2) Source Code Information: Section V-A motivates the need to add equalities between symbolic variables to show equivalence of path conditions from different program versions. To achieve this, we modified SPF to track this additional source code information.

We need to link the symbolic variable back to the corresponding point in the source code. However, symbolic PathFinder works on Java bytecode and does not maintain such information. We modify SPF to intercept and adapt the creation of symbolic variables. Upon creation of a symbolic variable, our modified Symbolic PathFinder saves any line number and original name information.

C. Constraint Solver

We use the Z3 constraint solver [12]. We extend SPF's built-in translation of path conditions to queries. Eventually, the queries are processable by Z3.

Passing constraints and path conditions to Z3, and to other constraint solvers, is already built into SPF. We extend the existing SPF implementation to fit our needs. We patched support for `implies` and `or`, which was missing, and support for `not`, which was limited.

To check for equivalence between path conditions, our tool aligns the links discovered by Gumtree in the AST with the source code information of the symbolic variables in SPF. Note that some discrepancies between line numbers can occur due to the Java compilation process. Our tool applies a heuristic looking for lines close enough, within a threshold of three lines.

D. Prototype

Our prototype combines the previous technical aspects. It performs a syntactical analysis of O, A, B, and M. It ensures that each version is symbolically executed. It finds equivalent path constraints. Finally, it checks whether any violations of CF-A, CF-B, or CF-AB occur.

Results are shown by listing path conditions, their equivalencies, and flagging those that break CF-A, CF-B, or CF-AB. A further improvement would be to link these path conditions back to the relevant parts of the code, for example, by using the source code information our prototype already keeps track of.

Our prototype requires a user to explicitly state the method and inputs of the program that need to be analysed symbolically. We also need to inject a `main` method into the code that is subject to analysis. This custom `main` guides symbolic execution to the relevant part of the code. We assume that future work can automate and resolve these limiting factors of our prototype.

VI. Evaluation

We evaluate our approach on synthetic and empirical data. The synthetic data is used to show the technical validity of our prototype. We use data generated by mutation testing. The empirical data is used to show the empirical relevance of our approach. This data is gathered from GitHub. The data and the prototype can be found online [1].

A. Technical Validation

This section evaluates the technical validity of our prototype on synthetic data.

1) Method: We generate a dataset using software mutation testing [13]. The original program version O is defined to be a Java program. We run the mutation testing tool Major [20] on the program to generate a set of mutations $\mathcal{U}$ of O. We form the Cartesian product of set $\mathcal{U}$ with itself, and define branches A and B accordingly. Finally, we use Git's default merge to produce merge M.

Most program versions O, A and B merge without problems. However, Git's textual merge potentially produces semantically conflicting program versions M. Our approach should detect such semantic conflicts.

For the resulting 4-tuples that we have generated, we manually classified the merge M as a conflict or not. We run our prototype and compare the output to this manually tagged baseline.

2) Results: For the input program O, we use code from *Project Euler*, a website centred around mathematical programming challenges [15].

We generate a set $\mathcal{U}$ with 34 different mutants of the original program. The Cartesian product results in 561 combinations for pairs of A and B with corresponding merge M. We exclude symmetric pairs. We also exclude 123 pairs being reported as a trivial textual conflict by Git's default merge. We randomly sample ten pairs from the remaining 438 for a manual analysis.

We manually classify 3 of the corresponding merges as semantic conflicts and 7 as valid. **Our tool classifies all merges correctly**.

3) Threats to Validity: The low complexity of the input program is a threat to validity for this technical evaluation. We used an existing program instead of creating an input program as part of this evaluation. This makes the results more realistic. However, the technical limitations, described in Sec. V, constrain the selection of our input program.

Our manual classification of semantic conflicts may be influenced by our understanding of our approach. This threat is hard to mitigate. We deploy our dataset online to enable the replication and revision of our approach.

The mutation generation is limited in that every mutation introduces exactly one change to the program. However, both changes affect the same parts of the program which is still a challenging situation.

B. Empirical Validation

The empirical evaluation of our approach is challenging because real semantic merge conflicts are rare. There is no standard dataset or way of defining them. Our empirical evaluation is thereby split into two parts.

- In a first part, we aim to classify semantic merge conflicts **retroactively**. We use heuristics on the revision history following a merge. We do this to get promising merge candidates that are interesting for an in-depth discussion. We need this alternative to a regular sample from GitHub, since semantic merge conflicts are rare. Manually tagging a rare class on a regular sample is unrealistic due to the vanishingly small number of positive (or negative) cases.
- In a second part, we apply our tool to such candidates and check if we can identify the conflicts **proactively** using our approach. This is done without having the subsequent commit history that indicates a conflict. We discuss such cases in-depth in TABLE I.

We structure this part of the evaluation accordingly.

[1] For review, the data and the prototype can be found online on https://figshare.com/s/44d8f650a07c1793321b

TABLE I

OVERVIEW OF THE CASES STUDIED IN RQ2

Real vs. Synthetic	Project	Merge	Description	Violated PCs …	
				at merge	after fix
R	google/j2objc	79781f8	In our first case, the branches A and B add the same behaviour. In branch A, the behaviour is added through a method call, protected by a boolean flag. In branch B, the behaviour is directly added to the method's body, and not protected by a boolean flag. The merge combines both modifications, which is a clear semantic conflict. In the fix, the code added in branch A is removed. This real-world example is comparable to our example in the introduction. However, the code of branch A includes a boolean flag, but the fix does not. Hence, our prototype still reports on some violations after the fix, but fewer violations compared to the merge M.	6 / 6	4 / 6
R	tcurdt/jdeb	e9ceff5	In the second case, branch A and B move code around and make some other changes. The result is code duplication in the merge M, and renders the code added by branch A as unreachable. The fix removes the unreachable code. Our tool warns about the behaviour from branch A disappearing for the merge M and after the fix. However, after the fix, the behaviour is still missing, so the number of violations does not change. This is a case where our prototype does not work as expected.	3 / 15	3 / 15
R	welovecoding/ editorconfig-netbeans	99578c4	In the third case, a bug is present in the origin O. In branch A, the method is fixed that contains the bug. In branch B, the call to the method is commented out. In merge M, the method was thus fixed, but not called. The fix uncomments the method call. Our tool reports about the missing changes from A disappearing. In the fixed version, the tool reports that the effect of B's commenting out of code has disappeared. The fix decreases the conflicts.	2 / 11	1 / 11
R	larsga/Duke	7c65f5e	In case four, the changes causing a semantic merge conflict involve a `try-catch` and the exception being raised within it. Our prototype cannot analyse this due to the technical limitations of Symbolic PathFinder that does not handle `try-catch`. We see no obvious way to simplify this case into something that can be analysed in the context of our prototype. We assume that our approach can spot this conflict if a `try-catch` is being supported by the Symbolic PathFinder.	-	-
R	spotify-web-api-java/ spotify-web-api-java	675a0d2	In case five, branch A modifies the method's return type. In branch B, a method with the same name is added, overloading the old method by adding a new parameter. The return type of this method in branch B is the same return type as in the original method before the change of branch A. In merge M, there is both, the method from A with the new return type and the method from B with both, the old return type and the added parameter. The fix updates the return type of the method of branch B to match the return type of branch A. This is something our approach is unable to spot. There are no path conditions to compare, and calling it a bug relies on guessing whether the developer intended to do have different return types or not.	-	-
R/S	google/j2objc	79781f8'	Case six is an adaptation of google/j2objc merge 79781f8 (our first case). The boolean flag that was added in version A of the original case, is not added here. Instead, the added method is always called. Thus, A and B behave entirely the same, while the behaviour is duplicated in M. The fix removes the method call from A. The fix behaves exactly like A and B. We detect no more violations after the fix.	2 / 2	0 / 2
S	-	-	Case seven is synthetic. Consider branch A and B, which add a `parse` and a `sanitise` function, respectively. The plan of the developers is that, once merged, `parse` will make use of `sanitise` to clean up its input. When merging, however, this call is not added to `parse`. Our tool reports no violations. In the fix, `parse` does call `sanitise`. Our tool reports a conflict: the behaviour of `parse` changes in M (the "fix" here) compared to its behaviour in branch A. We add this synthetic case to our dataset for our evaluation to show the limitation of our prototype.	0	not 0

1) Retroactive Method: We use a dataset of Java projects with Maven, introduced by Cavalcanti et al. [9]. The dataset lists merges, their parents, and metadata on the build and test success for the merge commits. However, Cavalcanti et al. study syntactic merge conflicts by comparing structured and semi-structured approaches to merging. Hence, this dataset does not immediately work for the evaluation of our approach. We use the same projects as Cavalcanti et al., but deviate in how we classify the merges.

For the Java projects, we collect all merges M, the corresponding parents A, B, and the first common ancestor O. We ignored rare situations where there are more than two parent commits. We also ignored trivial merges where A is a parent of B (or vice versa). To identify O, we use Git's built-in common base finding algorithm.

We apply the following heuristics to filter for interesting candidates that can potentially be classified as a semantic merge conflict.

- We filter for **merges with an overlap** in the modifications. We apply stratification (or group-by) to diversify. For one half of the merges, we require that versions A and B change the same file. For the other half, we require that the same line was changed.
- We filter for merges with a suspicious commit history following the merge. We apply a light-weight version of the SZZ algorithm [34] (also used in [23], [32]) to **classify the commit after the merge as a bug fix**. We do not go for the alternative of considering bug fixing commits further down in the tree of descendants. In previous experiments, we noticed that this introduces noise both due to an overlap between merges and due to a general increase in the number of unrelated fixes.
- The bug fix following the merge does not always relate to a bug introduced in the merge. Hence, the changes by the **fix and the merge need to overlap**, too. We filter for bug fixing commit that at least change one line or file (see earlier discussion on stratifications) of the lines or files changed between versions O and M.

The heuristics are relevant to reduce the candidates subject to our subsequent manual tagging of conflicts. They produce a pool of candidates with a realistic chance of being a semantic merge conflict. Executing a manual analysis on a random sample of merges on GitHub would be unrealistic. Our method corresponds to a standard practice in evaluating information retrieval systems, referred to as *relevance judgement* or *pooling method* (see [36], page 13, or more recently [2], page 158).

2) Retroactive Results: We manually examined 500 merges in 152 projects identified by our retroactive method. We identified 55 semantic merge conflicts, where 50 were caught by the compiler. The five remaining cases are the interesting ones for our approach. The conflicts can be found in TABLE I.

All five conflicts were identified by the heuristic of overlapping changes to files (not to lines). All five conflicts include a bug fixing commit message using the keyword `fix`. Two of the messages also use the keyword `merge`.

3) Proactive Method: In the following part, we evaluate our proactive approach using symbolic execution empirically by running it on the pool of candidates that we have identified in the previous part of the evaluation.

To get a more exhaustive discussion, we complement the candidates by i) one adaptation that simplifies the control flow and ii) another synthetic case where our approach does not work as expected.

4) Proactive Results: The results are described in Table I. The table does not include the merges that we have manually tagged as conflict free. We structure the table as follows:

- The first column (**Real vs. Synthetic**) indicates a discussion of a real-world or synthetic case. The majority of the cases are real. Synthetic cases are added to make the discussion exhaustive.
- The second column (**Project**) reports on the GitHub project name.
- The third column (**Merge**) lists the abbreviated SHA of the merge.
- Column four (**Description**) explains the semantic conflict in depth and discusses the pros and cons of the classification done by our approach.
- Column five and six (**Violated PCs at the merge or after the fix**) give the actual classification in terms of violated path conditions. This number is either given for the merge or for after the fix. Eventually, we expect this number to decrease after a fix if our approach works correctly. The number of violations can be derived from our definition in set semantics (see IV-D).

To sum up the insight of the table: **Our approach using symbolic execution correctly detects the decrease in violation (by fixes) for three merges. The violations of one merge disappear (by the fix). For one merge, our approach reports on no change in violation (by the fix). In one synthetic case, violations increase (by the fix).** Two merges cannot be processed due to technical limitations.

This shows that in specific cases, our approach using symbolic execution is a promising extension to existing mechanisms to merge conflict detection. However, there are some cases for which the approach does not work as expected. These are detailed in the table.

5) Threats to Validity: Comparable to the technical part of this evaluation section, our manual classification of merge conflicts may be biased by our expectations. We deploy our dataset online to allow replication and revisions of our approach.

Different to the technical part of the evaluation that uses mutations, we cannot make any objective statements on the developer's original intention in the real-world cases. We try to mitigate this problem by manually inspecting all involved commits and the commit messages in depth.

While we examined 500 merges, the low size of positive cases is another threat to this analysis and illustrates the rare nature of semantic merge conflicts. We are not computing any confidence intervals for which such sample size matters. We try to mitigate the problem of a low number of positives by favouring a qualitative and exhaustive discussion of real cases, including synthetic cases that logically follow from the real cases we have spotted.

VII. Conclusion

In this paper, we develop an approach that detects semantic merge conflicts by symbolic execution. We define the program semantics as path conditions, produced by a symbolic executor, and check whether the conditions satisfy established rules that reflect a merge conflict. Our usage of symbolic execution to check these rules is novel. We implement a prototype for the evaluation of our approach.

We evaluate the technical validity of our prototype using synthetic data. We generate the synthetic data through mutation testing.

We evaluate our approach empirically by following our research questions:

- **RQ1**: We need empirical data from GitHub to show the empirical relevance of our approach, but semantic merge conflicts are rare and hard to spot manually. Hence, we define a retroactive method to do an approximate classification of semantic merge conflicts that scales. The method is based on the commit history following the merge. We use the method to compute a pool of potential semantic merge conflict, which we examine manually and use for answering RQ2.
- **RQ2**: We use the pool of manually classified semantic merge conflicts to evaluate our approach to detect semantic merge conflicts by symbolic execution. We discuss the application of our prototype qualitatively for five cases from our pool of real-world semantic merge conflicts.

Our approach using symbolic execution correctly detects changes in semantic violations in three out of five semantic merge conflicts. The evaluation shows that in specific cases, our approach using symbolic execution is a promising extension to existing mechanisms to merge conflict detection.

Our prototype is limited within the technical constraints of the symbolic executor we used. Future work needs to focus on automation and the integration of the approach into the continuous integration pipeline. We also consider the combination of symbolic execution and data-driven approaches, such as deep-learning, as promising future work. This may overcome validity related limitations of data-driven approach by a more formal approach to symbolic execution.

References

[1] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, “Semistructured merge: rethinking merge in revision control systems,” in *19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) and 13th European Software Engineering Conference (ESEC)*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 190–200.

[2] R. Baeza-Yates and B. Ribeiro-Neto, *Modern information retrieval*. Pearson Education, 2011, vol. 463.

[3] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. Germán, and P. T. Devanbu, “Cohesive and isolated development with branches,” in *FASE*, ser. Lecture Notes in Computer Science, vol. 7212. Springer, 2012, pp. 316–331.

[4] C. Brindescu, I. Ahmed, C. Jensen, and A. Sarma, “An empirical investigation into merge conflicts and their effect on software quality,” *Empirical Software Engineering*, vol. 25, no. 1, pp. 562–590, sep 2019.

[5] C. Brindescu, I. Ahmed, R. Leano, and A. Sarma, “Planning for untangling: Predicting the difficulty of merge conflicts,” in *ICSE 2020*, 2020.

[6] 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)*, 2011.

[7] J. Buffenbarger, “Syntactic software merging,” in *Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Selected Papers*, ser. Lecture Notes in Computer Science, J. Estublier, Ed., vol. 1005. Springer, 1995, pp. 153–172.

[8] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, February 2013.

[9] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, “The impact of structure on software merging: Semistructured versus structured merge,” in *Automated Software Engineering (ASE)*, 2019.

[10] L. Da Silva, P. Borba, W. Mahmood, T. Berger, and J. Moisakis, “Detecting semantic conflicts via automated behavior change detection,” in *International Conference on Software Maintenance and Evolution (ICSME)*, 2020.

[11] B. de Alwis and J. Sillito, “Why are software projects moving from centralized to decentralized version control systems?” in *CHASE*. IEEE Computer Society, 2009, pp. 36–39.

[12] L. de Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” *Computer*, 1978.

[14] E. Dinella, T. Mytkowicz, A. Svyatkovskiy, C. Bird, M. Naik, and S. K. Lahiri, “Deepmerge: Learning to merge programs,” *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 1599–1614, 2023.

[15] P. Euler. Problem 1: Multiples of 3 or 5. [Online]. Available: https://projecteuler.net/problem=1

[16] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” in *ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds. ACM, 2014, pp. 313–324.

[17] K. Havelund and T. Pressburger, “Model checking java programs using java pathfinder,” *International Journal on Software Tools for Technology Transfer*, 2000.

[18] S. L. Herrera Gonzalez and P. Fraternali, “Almost rerere: Learning to resolve conflicts in distributed projects,” *IEEE Transactions on Software Engineering*, pp. 1–18, 2022.

[19] S. Horwitz, J. Prins, and T. Reps, “Integrating noninterfering versions of programs,” *ACM Transactions on Programming Languages and Systems*, 1989.

[20] R. Just, “The major mutation framework: efficient and scalable mutation analysis for java,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, 2014.

[21] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, 1976.

[22] T. Mens, “A state-of-the-art survey on software merging,” *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, May 2002.

[23] A. Mockus and L. G. Votta, “Identifying reasons for software changes using historic databases,” in *International Conference on Software Maintenance (ICSM)*, 2000.

[24] W. Muylaert and C. De Roover, “Prevalence of botched code integrations,” in *International Conference on Mining Software Repositories (MSR)*, 2017.

[25] N. Nelson, C. Brindescu, S. McKee, A. Sarma, and D. Dig, “The lifecycle of merge conflicts: processes, barriers, and strategies,” *Empirical Software Engineering*, vol. 24, no. 5, pp. 2863–2906, feb 2019.

[26] Y. Noller, C. Pasareanu, M. Bohme, Y. Sun, H. L. Nguyen, and L. Grunske, “Hydiff: Hybrid differential software analysis,” in *ICSE 2020*, 2020.

[27] R. Pan, V. Le, N. Nagappan, S. Gulwani, S. Lahiri, and M. Kaufman, “Can program synthesis be used to learn merge conflict resolutions? an empirical analysis,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, may 2021.

[28] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[29] 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, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 570–581.

[30] S. Phillips, J. Sillito, and R. Walker, "Branching and merging: An investigation into current version control practices," in *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2011.

[31] R. Premraj, A. Tang, N. Linssen, H. Geraats, and H. van Vliet, "To branch or not to branch?" in *International Conference on Software and Systems Process (ICSSP)*, 2011.

[32] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *International Conference on Software Engineering (ICSE)*. ACM Press, 2016.

[33] E. Shihab, C. Bird, and T. Zimmermann, "The effect of branching strategies on software quality," in *2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2012.

[34] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes? (on fridays.)," in *Mining Software Repositories (MSR)*, 2005.

[35] M. Sousa, I. Dillig, and S. K. Lahiri, "Verified three-way program merge," *PACMPL*, vol. 2, no. OOPSLA, pp. 165:1–165:29, 2018.

[36] K. Sparck-Jones and C. J. V. Rijshergen, "Report on the need for and provision of an "ideal" information retrieval test collection," 1975.

[37] C. Sung, S. K. Lahiri, M. Kaufman, P. Choudhury, and C. Wang, "Towards understanding and fixing upstream merge induced conflicts in divergent forks: An industrial case study," in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020.

[38] A. Svyatkovskiy, S. Fakhoury, N. Ghorbani, T. Mytkowicz, E. Dinella, C. Bird, J. Jang, N. Sundaresan, and S. K. Lahiri, "Program merge conflict resolution via neural transformers," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, nov 2022.

[39] S. S. Towqir, B. Shen, M. A. Gulzar, and N. Meng, "Detecting build conflicts in software merge for java programs via static analysis," in *International Conference on Automated Software Engineering (ASE)*, 2022.

[40] J. Tsay, L. Dabbish, and J. D. Herbsleb, "Influence of social and technical factors for evaluating contribution in github," in *ICSE*. ACM, 2014, pp. 356–366.

[41] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Proceedings of the 3rd International Workshop on Software Configuration Management, Trondheim, Norway, June 12-14, 1991*, P. H. Feiler, Ed. ACM Press, 1991, pp. 68–79.

[42] T. Wuensche, A. Andrzejak, and S. Schwedes, "Detecting higher-order merge conflicts in large software projects," in *International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, oct 2020.