Detecting Build Conflicts in Software Merge for Java Programs via Static Analysis

Sheikh Shadab Towqir
Virginia Tech
USA
shadabtowqir@vt.edu

Bowen Shen
Virginia Tech
USA
bowenshe@vt.edu

Muhammad Ali Gulzar
Virginia Tech
USA
gulzar@vt.edu

Na Meng
Virginia Tech
USA
nm8247@vt.edu

ABSTRACT

In software merge, the edits from different branches can textually overlap (i.e., *textual conflicts*) or cause build and test errors (i.e., *build* and *test conflicts*), jeopardizing programmer productivity and software quality. Existing tools primarily focus on textual conflicts; few tools detect higher-order conflicts (i.e., build and test conflicts). However, existing detectors of build conflicts are limited. Due to their heavy usage of automatic build, current detectors (e.g., Crystal) only report build errors instead of identifying the root causes; developers have to manually locate conflicting edits. These detectors only help when the branches-to-merge have no textual conflict.

We present a new static analysis-based approach BUCOND ("build conflict detector"). Given three code versions in a merging scenario: base b, left l, and right r, BUCOND models each version as a graph, and compares graphs to extract entity-related edits (e.g., class renaming) in l and r. We believe that build conflicts occur when certain edits are co-applied to related entities between branches. BUCOND realizes this insight via pattern matching to identify any cross-branch edit combination that can trigger build conflicts (e.g., one branch adds a reference to field F while the other branch removes F). We systematically explored and devised 57 patterns, covering 97% of the build conflicts in our experiments. Our evaluation shows BUCOND to complement build-based detectors, as it (1) detects conflicts with 100% precision and 88%–100% recall, (2) locates conflicting edits, and (3) works well when those detectors do not.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; Collaboration in software development.

KEYWORDS

software merge, build conflicts, static analysis, pattern matching

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3556950

ACM Reference Format:
Sheikh Shadab Towqir, Bowen Shen, Muhammad Ali Gulzar, and Na Meng. 2022. Detecting Build Conflicts in Software Merge for Java Programs via Static Analysis. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556950

1 INTRODUCTION

Developers create software branches for tentative feature addition or bug fixing. They periodically integrate (i.e., *merge*) code changes from distinct branches to release software with new features or patches. In practice, the merge process is rarely straightforward due to *conflicts*, i.e., the conflicting edits simultaneously applied in branches-to-merge. Developers often spend hours or days detecting and resolving conflicts before correctly merging branches [42].

A typical **merging scenario** in software version history involves four program commits: the base ***b***, left version ***l***, right version ***r***, and developers' merge result ***m*** (see Figure 1). Between l and r, there can be three types of merge conflicts [33, 59]: textual, build, and test conflicts. **Textual conflicts** are caused by divergent branch edits to the same line(s) of text, while **higher-order conflicts** (i.e., build and test conflicts) are caused by edits simultaneously applied to different lines. In particular, **build conflicts** produce build failures when m is compiled. **Test conflicts** trigger test errors when m compiles successfully and gets executed with test cases.

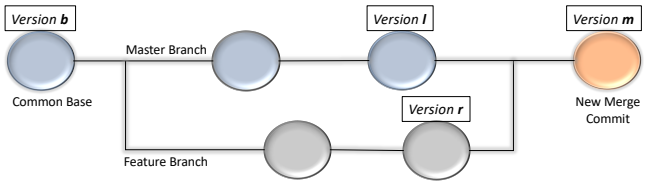

Figure 1: An exemplar merging scenario

Table 1: Existing tool support for conflict detection

	Textual conflicts	Build conflicts	Test conflicts
Tools	git-merge, FSTMerge, AutoMerge JDime, AutoMerge, IntelliMerge, Crystal, WeCode	Crystal, WeCode, IntelliMerge	Crystal, WeCode, SafeMerge

Existing tools offer limited support for conflict detection in Java programs [6, 28, 29, 33–35, 40, 44, 48, 56, 57, 59, 62]. As shown in Table 1, majority of the tools target textual conflicts. Crystal [33] and WeCode [40] are among the few tools that detect all types of conflicts. They apply textual-merge of version control systems (e.g.,

Changes in *l*	Changes in *r*
public class A {	public class A {
public void foo() {	public void foo() {
+ C.m();	...
... } } ...	} } ...
public class C {	public class C {
public static void m() {...}	- public static void m() {...}
	+ public static void m(int p) {...}
}	}

Figure 2: An exemplar build conflict

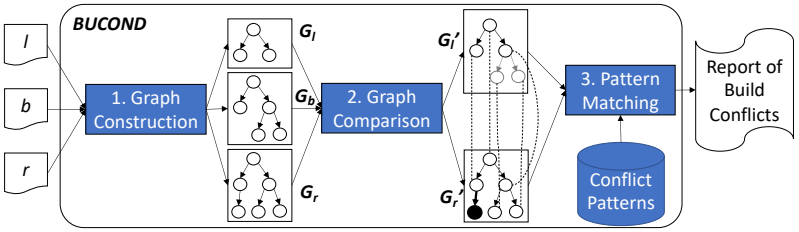

Figure 3: BUCOND comprises three phases

git-merge) to tentatively merge two branches into one version A_m, revealing textual conflicts along the way. Notice that developers often create m based on A_m, so A_m can be different from m for two reasons: (1) it shows all detected textual conflicts for developers to resolve; (2) it may have build or test errors that developers should fix to create m. If A_m does not show any textual conflict, Crystal and WeCode use automatic build to compile A_m. If l and r compile successfully but A_m fails, there are build conflicts between branches. Lastly, the tools test the compiled version of A_m. If the compiled versions of l and r pass all tests but A_m fails any test, there are test conflicts between branches.

The compiler-based (i.e., build-based) detectors of build conflicts have three limitations. First, when textual conflicts coexist with build conflicts, A_m marks all detected textual conflicts with special lines ''<<<<<<< Head'', ''======='', and ''>>>>>>>...'' in code. Such program versions do not compile; thus, the automatic build process is not runnable to reveal any error. Second, given a build error in A_m, developers must manually locate the conflicting edits responsible for that error. This manual localization process can be challenging and time-consuming, especially when the symptom of error is geographically separated from the the root cause. Third, during the build process, the build errors found earlier can prevent compilers from detecting subsequent errors. In such cases, developers must resort to multiple iterations of automatic build, manual diagnosis of root causes, and manual conflict resolution to expose all errors. Because manual conflict resolution can take hours or days [42], such an iterative process can be very tedious and error-prone.

To overcome all limitations mentioned above, we created BUCOND, a new approach that detects build conflicts in Java code using static analysis. Our approach is based on two insights. First, *build conflicts often occur when cross-branch edit combinations violate the def-use constraints between program entities.* We use **program entities** to refer to Java program components like classes, methods, or fields. Please refer to Table 3 (Section 3) for the complete list of program entities. Figure 2 shows an exemplar build conflict. A conflict exists because l adds a call to `m()` while r updates the method signature. The co-application of both edits can produce an unresolved method reference. Second, *due to the limited number of entity types in Java code and limited edit types applicable to entities, it is possible to enumerate all cross-branch edit combinations that violate def-use constraints.* By defining conflict patterns for such combinations, we can compare the co-applied edits between branches to report a conflict whenever a pattern is matched.

BUCOND has three phases. As shown in Figure 3, given the three program versions of a merging scenario: b, l, and r, BUCOND creates a program entity graph (PEG) for each version, to model entities (e.g., Java methods) and inter-entity relations (e.g., method call). Phase II compares the PEGs of l and b, and compares the PEGs of r and b, to extract entity-related edits in both branches. It embeds all edit information in respective PEGs, creating new PEGs G_l' and G_r'. Our systematic enumeration revealed 57 types of cross-branch edit combinations that can cause build errors in the merged software. Accordingly, we defined 57 patterns and implemented 57 matchers in BUCOND. Those matchers are used to locate conflicting edits between branches. For each pattern, Phase III searches among edits in G_l' and G_r', and reports conflicts when matches are found.

We evaluated BUCOND with 3 datasets: (1) 57 merging scenarios with in total 57 synthetic conflicts, (2) 55 scenarios with in total 81 real conflicts that trigger build errors in A_m, and (3) 13 scenarios with 17 real conflicts that coexist with textual conflicts and got located by us manually. On Dataset 1, BUCOND detected all conflicts accurately. On Dataset 2, it detected conflicts with 100% precision, 95% recall, and 97% F-score. On Dataset 3, BUCOND achieved 100% precision, 88% recall, and 94% F-score. BUCOND complements compiler-based detectors for three reasons. First, it detects conflicts with high precision and high recall. Second, it pinpoints the root causes of build conflicts, while compiler-based tools only present the symptoms (i.e., build errors). Third, BUCOND detects conflicts via static analysis instead of automatic build; therefore, it helps reveal conflicts when compiler-based tools are inapplicable (i.e., textual and build conflicts coexist). Our research will help developers merge software more effectively and efficiently. We open-sourced our program and data at https://figshare.com/s/459145063f38bdb244b9.

2 A PRELIMINARY STUDY

Before our approach design, we conducted a pilot study to understand how build conflicts occur. To make our study representative, we randomly picked eight popular Java repositories on GitHub: fastjson [13], spring-cloud-alibaba [21], druid [11], redisson [20], litemall [16], mybatis-plus [17], javapoet [14], and jedis [15]. We chose these repositories because they are popular (i.e., with 9.5K–25.4K stars and 1.2K–8.1K forks) and from different domains.

In each selected repository, we searched for merging scenarios, i.e., any commit with two parent/predecessor commits. We use l and r to refer to the two parent commits in sequence. We treat the common child and ancestor commits between l and r as m and b. For each scenario, we first applied git-merge to l and r to generate a text-based merge version A_m. If A_m had no textual conflict, we further built l, r, and A_m. If l and r built successfully but A_m did not, there are build conflicts between l and r. To locate those conflicting edits, we analyzed the reported build errors, manually related those errors with program differences among all five relevant program versions (l, r, b, m, A_m), and identified the integrated branch edits responsible for those errors.

Table 2: Classification of the 25 build conflicts in our preliminary study based on their root causes

Idx	Conflict Type	Description	# of Conflicts
1	Import: remove def vs. add use	One branch removes a class import from a Java file, while the other branch adds reference(s) to that class.	3
2	Class: remove def vs. add use	One branch removes the definition of a Java class, while the other branch adds reference(s) to that class.	3
3	Class: add method def in super vs. add sub class	One branch adds a method M in a class A, while the other branch adds a class B to extend A. There is a method in B, whose method name and parameter list are identical to that of M but the return type is different. Namely, the return types between super and sub methods conflict.	1
4	Interface: change a class to implement the interface vs. change a method's return type in the class	One branch updates a class B to implement interface A. The other branch changes the return type of a method M in class B. The return types between the super and sub versions of M conflict.	9
5	Method: change the parameter list vs. add use	One branch changes the parameter list of a Java method, while the other branch adds reference(s) to the original method signature.	5
6	Field: remove def vs. add use	One branch removes the definition of a Java field, while the other branch adds reference(s) to that field.	3
7	Field: change a field's type vs. add write access	One branch updates the data type of a field, while the other branch adds reference(s) to that field based on the old data type.	1

As shown in Table 2, our study revealed 25 build conflicts, which were classified into 7 types based on the edited entities and edit types. For instance, three conflicts are of Type-1; they occur when one branch removes a class import and the other branch adds reference(s) to that originally imported class. Four conflicts are about fields (i.e., Type-6 and Type-7). They happen because one branch removes or updates a field `F` and the other branch adds reference(s) to the original field. Ten conflicts were concerning methods in super-sub types (i.e., Type-3 and Type-4). Namely, when a sub-class is edited to inherit a super-class or implement an interface, the methods defined in the super- and sub-types should not conflict. In other words, if both super- and sub- classes define a method with the same signature but different return types, automatic build fails.

Although the inspected conflicts are from distinct program contexts and have different root causes, they all convey the same message: **build conflicts can occur when cross-branch edit combinations violate the def-use constraints between program entities**. Frequently applied edits involve additions, deletions, and updates of entities' defs/uses; typical def-use constraints include:

(1) When an entity is referenced, there should always be a corresponding entity definition visible to the reference.
(2) No entity should be defined multiple times, except for method overriding.
(3) When a sub-class implements an interface, the class should implement all methods declared by the interface.
(4) When a sub-class implements or overrides a method M declared by a super-type, the sub-class should use the name, parameter list, and return type of M in its method definition.

Our study implies that if we can characterize the types of branch edits whose combination violates any def-use constraint, we do not need to wait for developers to produce A_m or to use automatic build for conflict detection. Instead, we can conduct static analysis to eagerly relate edits simultaneously applied to distinct branches, reason about the semantics of edits, and notify developers of potential conflicts before they actually merge software.

3 APPROACH

Inspired by our preliminary study, we designed and implemented BUCOND (short for "build conflict detector"), a novel approach to detect build conflicts via static analysis. In our research, we need to tackle two technical challenges:

C1. How can we derive entity-related edits from l and r?
C2. How can we relate edits across branches to identify conflicts?

To address these challenges, we designed a three-phase approach. As shown in Figure 3, Phases I and II create and compare graphs to extract entity-related edits, addressing C1. For C2, we defined a pattern set of conflicting edits in Phase III, based on our systematic exploration of potential conflict scenarios. With those patterns defined, Phase III detects conflicts via pattern matching in graphs. Sections 3.1–3.3 explain all phases in detail.

3.1 Phase I: Graph Construction

Our research intends to detect conflicts by extracting and contrasting the entity-related edits of each branch. However, the default program diff information recorded in software repositories does not serve that purpose for two reasons. First, the program diff of l or r records the changes each branch applied to the base b, instead of the differences between l and r. More importantly, many of the recorded changes are irrelevant to any entity's def or use (e.g., adding an `if`-statement), and should be omitted for efficient static analysis. Second, to identify potential conflicts between branch edits, we need to relate applied edits with their surrounding context (i.e., unchanged code). Program diff shows applied edits but provides insufficient contextual information for conflict recognition.

To facilitate the extraction and comparison of edits, BUCOND creates a **program entity graph (PEG)** separately for b, l, and r. Specifically, given two program commits to merge in a Git repository, c_l and c_r, BUCOND applies the command "`git merge-base`" to retrieve the common base commit c_b. Next, BUCOND locates all edited Java files by c_l or c_r, and creates three folders to separately hold the base, left, and right versions of those files. For instance, if a file is updated by either branch, its three versions are put into separate folders. If a file is added by a branch, its unique version is only put into the branch's corresponding folder. Notice that BUCOND only scans versions of **edited files** (i.e., added, deleted, updated, renamed, or moved files) when modeling PEGs. This is because a commit often edits a small portion of files [31, 41, 61]. If we include all Java files into graph modeling, the resulting graphs can become unnecessarily huge. Meanwhile, we noticed that when l and r build successfully, the edited files always contain all edit-related details. Thus, it is safe to only analyze edited files to detect build conflicts.

BUCOND traverses each folder, and parses every source file with JavaParser [5] to create abstract syntax trees (ASTs). Based on the ASTs, BUCOND extracts entities as well as inter-entity relations, and uses JGraphT [49] to build PEGs. In each PEG, *vertices* represent *entities* and *edges* show *inter-entity relations*. Figure 4 shows three exemplar PEGs for the merging scenario of Figure 2. In Figure 4, each node records both the type and fully qualified name of an

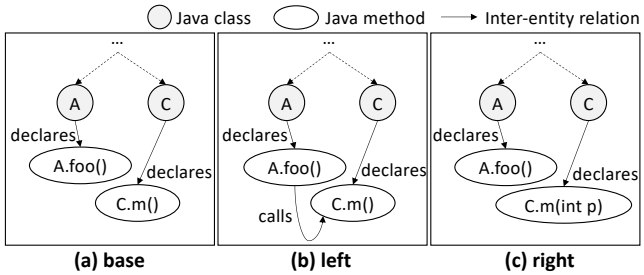

Figure 4: The PEGs for the merging scenario in Figure 2

Table 3: Types of vertices and edges in a PEG

Types of Source Vertices/Entities	Types of Possible Outgoing Edges	Types of Target Vertices
Project (prj)	contains	pkg
Package (pkg)	contains	cu
Compilation Unit (cu)	imports	pkg, cls, itf, enm
	declares	cls, itf, enm
Class (cls)	extends	cls
	implements	itf
	declares	fld, mtd, ctr, cls
Interface (itf)	declares	fld, mtd
Enum (enm)	declares	fld, ctr, ec
Field (fld)	reads	fld, ec
	calls	mtd, ctr
	initializes	cls
Method (mtd)	reads	fld, ec
	writes	fld
	calls	mtd, ctr
	initializes	cls
Constructor (ctr)	reads	fld, ec
	writes	fld
	calls	mtd, ctr
	initializes	cls
Enum Constant (ec)	-	-

"-" means zero entry

entity (e.g., `A.foo()`). Each edge is labeled with the relation type (e.g., "declares"). There are 10 node types and 9 edge types in PEGs (see Table 3). Specifically given entities E_1 and E_2,

- E_1 **contains** E_2 means the file folder of E_1 includes the file (or folder) of E_2.
- E_1 **imports** E_2 means a compilation unit E_1 imports E_2, where E_2 is a package, a class, an interface, or an enum type.
- E_1 **declares** E_2 means E_1 declares another entity E_2 inside its implementation. For instance, "cls declares cls" means "a class declares an inner class".
- E_1 **extends** E_2 means a type (i.e., class or interface) E_1 inherits its fields and methods from another type E_2. We use "**sub**" and "**super**" to refer to E_1 and E_2.
- E_1 **implements** E_2 means a class E_1 implements an interface E_2. In such scenarios, we also use "sub" and "super" to separately refer to E_1 and E_2.
- E_1 **reads** E_2 means E_1 references E_2 for its value. For example, "mtd reads fld" means "a method reads a field's value".
- E_1 **writes** E_2 means that E_1 references E_2 to store a value to E_2. For instance, "cts writes fld" means "a constructor writes a value to a field".
- E_1 **calls** E_2 means the definition of E_1 calls a function (i.e., a method or constructor) E_2. For instance, "fld calls mtd" means "the definition statement of a field calls a method".
- E_1 **initializes** E_2 means the definition of E_1 calls a constructor of class E_2.

Among the nine relations, "contains" and "declares" serve as ways to define E_2. "Imports" can be considered as both def and use of E_2, because an import declaration uses an entity defined by another file and defines the imported entity E_2 for the current file. The other six relations show alternative ways to use entity E_2. We intentionally differentiated between the read and write accesses of entities, as in certain scenarios (e.g., a `final` field) we handle these accesses differently (see Section 3.3). Additionally, we modeled two edges for each constructor invocation: "entity calls ctr" and "entity initializes cls". This is because constructors are different from general Java methods in three ways. First, they share names with the declaring classes. Second, even if a class A defines no constructor, the default implicit constructor with no argument $A()$ is always callable. Third, any explicitly defined constructor replaces such an implicit constructor. By tracking the relations of any constructor caller with (1) the constructor declaration and (2) the declaring class, we can comprehensively relate edits with their context.

Algorithm 1: Graph construction

```
   Input  : F, /* list of edited files for a given branch */
   Output : G, /* constructed PEG for a given branch */
1.1  G ← ∅; /* PEG to store a set of nodes and edges */
     /* Step 1: Traverse each AST to extract all entities, and add nodes as well as
        related contains/declares/imports edges to G. */
1.2  foreach f ∈ F do
1.3      ast ← parseAST(f); /* parse each Java file */
1.4      traverse(ast, G);
     /* Step 2: Enumerate all entity nodes, map imported entities, and add the
        other six types of edges as needed. */
1.5  foreach n ∈ G do
1.6      if n.nodeType == cu then
1.7          mapImports(n, G);
1.8      else if n.nodeType == cls then
1.9          addExtendImpls(n, G);
1.10     else
1.11         // add reads/writes/calls/initializes edges as needed
             addOtherEdges(n, G);
```

Algorithm 1 overviews our procedure of graph construction. This algorithm consists of two steps. As shown by lines 1.2–1.4, *Step 1* parses each edited Java file in a given branch to create ASTs. It also traverses ASTs to extract entities as well as `contains/declares/imports`-relations between entities, in order to add nodes and edges to G. Specifically, `contains`-edges are created based on the package declarations in individual Java files; `declares`-edges are created based on the parent-child relations between entities in ASTs; `imports`-edges are created based on the import declarations of each compilation unit. Here, for each imported entity, BUCOND creates a dummy node to hold the entity name, because *Step 2* will map some of the entities to their actual nodes parsed from Java files.

Note that this step does not attempt to extract the other six types of relations (e.g., `reads`) or add edges for those relations. The reason is that before adding all nodes to G, BUCOND cannot always locate the target nodes of potential edges within the analysis scope. To facilitate later addition of edges, this step also stores necessary information into nodes, including the types and fully qualified names of nodes, `extends/implements`-related info for classes, `read/write`-accesses for fields, and function calls by the statements of field/method/constructor declarations.

Step 2 enumerates all nodes in G, to map dummy `imports`-targets to nodes extracted via AST traversal, and to add extra edges based

on the information stored by *Step 1* (lines 1.5–1.11). Specifically, if a given node n is a compilation unit, BUCOND scans the `imports`-related dummy nodes created by *Step 1*, and maps those nodes to nodes actually extracted from edited Java files. If an imported entity e is not defined by any analyzed Java file, e.g., e is a class defined by JDK or a third-party library, BUCOND keeps the dummy node as a placeholder for e's declaration. Otherwise, if e is defined by an analyzed Java file, BUCOND connects the dummy node with e's node via an `imports`-edge. Such special handling is due to the dual role played by an import declaration: it uses an entity defined elsewhere and defines an imported entity locally.

Alternatively, if n is a class, BUCOND scans the `extends/implements`-related info to locate nodes corresponding to the extended class and implemented interfaces, and adds edges accordingly. If n is a field, method, or constructor, its declaration body may access fields or call functions (i.e., methods or constructors). BUCOND scans related AST nodes to add `reads/writes/calls/initializes` edges. BUCOND adopts the built-in JavaSymbolSolver of JavaParser to resolve type bindings for names of called methods, and uses string matching to tentatively resolve type bindings for field accesses.

When JavaSymbolSolver fails to resolve bindings for some method calls like `m(...)`, BUCOND implements a naïve approach to search for any entity defined with that name, to recognize the inter-entity relation. Namely, BUCOND searches for all methods defined with `m`, and tentatively compares the methods' parameter types as well as parameter counts with that of `m`-call. If only one method matches the method call, BUCOND considers this method's node as the `calls`-target. Otherwise, if multiple methods can match the call, BUCOND does not link the caller to any method's node for conservativeness.

3.2 Phase II: Graph Comparison

We refer to the PEGs created for distinct program versions with the following notations: G_b, G_l, and G_r. This phase compares G_l and G_r separately with G_b, to derive entity-related edits for each branch. The phase has three steps: content-based matching, similarity-based matching, and edit generation.

3.2.1 Content-Based Matching. When comparing two graphs, BUCOND first matches entity nodes purely based on their content. Namely, for each node, BUCOND computes a unique ID—a hashcode of the node type and fully qualified name (FQN). It then compares hashcodes across graphs to match nodes. All node matches are then recorded in a map $\boldsymbol{M}$. For the PEGs in Figure 4, the comparison between G_b and G_l results in a complete match between nodes, as l did not modify any entity's FQN. Meanwhile, the comparison between G_b and G_r only reveals three pairs of node matches; it cannot match the nodes of $C.m(\ldots)$ across graphs because the right branch r updated the method signature.

3.2.2 Similarity-Based Matching. For all unmatched nodes between two graphs, BUCOND further sorts nodes based on their types, and compares same-typed nodes by their surrounding context. Specifically for a node n, we use **context** to refer to the nodes that are directly connected with n via edges. Given two same-typed nodes n_1 and n_2 and their contextual node sets N_1 and N_2, we compute the similarity as below:

$$Context\_Sim = \frac{N_1 \cap N_2}{N_1 \cup N_2} \quad (1)$$

Here both the set intersection and union are computed based on the node matches recorded in M. *Context_Sim* varies within [0, 1]. The higher this value is, the more similar n_1 is to n_2.

For most node types (except methods, constructors, fields, and enum constants), BUCOND uses contextual similarity to decide how similar two given nodes are to each other. If the score is above a threshold (i.e., the golden ratio 0.618 [3]), we consider the two nodes similar enough to match. We chose 0.618, because it is used by prior work [56] and led to reasonably good results. When a node from a graph successfully matches multiple nodes in the other graph, BUCOND picks the one with the highest similarity score.

For the remaining four node types (i.e., methods, constructors, fields, and enum constants), contextual similarity is insufficient to match nodes accurately for two reasons. First, *Context_Sim* cannot easily differentiate between entities within the same context. For instance, when multiple fields are located in the same class and all initialized with `null`, they have identical context. Second, in addition to FQNs, these entities also have separate code implementation, such as statements inside a method body or expressions inside an enum constant (i.e., `public enum Planet{Mercury(3.303e+23, 2.4397e6), ...}`). Such code implementation can help further differentiate the same-context entities. Therefore, we defined three additional formulas to compute the similarity scores for the four entity types:

$$Function\_Sim = (Name\_Sim + Context\_Sim + Body\_Sim)/3 \quad (2)$$

$$Field\_Sim = (Name\_Sim + Type\_Sim + Expression\_Sim)/3 \quad (3)$$

$$EC\_Sim = (Name\_Sim + Context\_Sim + Expression\_Sim)/3 \quad (4)$$

Formula (2) computes the similarity between method (or constructor) nodes as the mean value of *Name_Sim*, *Context_Sim*, and *Body_Sim*. Here, *Name_Sim* is the **string similarity** derived from the n-grams of both method names, where $n = 3$. Here, we set $n = 3$ because the setting is used by prior work [56] and shows great effectiveness in experiments.

Body_Sim describes how similar two method (or constructor) bodies are to each other. We reused GumTree [39] to compute the AST similarity between methods. GumTree takes in two ASTs, and uses a greedy top-down algorithm as well as a bottom-up algorithm to map AST nodes. Once all mappings are established, BUCOND computes the similarity score by dividing the total number of node matches with the node count of the larger AST.

Formula (3) computes the similarity between field nodes as the mean value of *Name_Sim*, *Type_Sim*, and *Expression_Sim*. Different from functions, each field declaration consists of only one statement with the typical format "`Type fieldName [ = Expression]`". We cannot naïvely compare the ASTs of fields to measure similarity, as any minor difference in these ASTs can significantly impact the measured value. Instead, we compute the string similarities of (1) type names, (2) field names, and (3) (optionally) expressions, and average them for the final result.

Formula (4) computes the similarity between enum constants as the mean value of *Name_Sim*, *Context_Sim*, and *Body_Sim*. An enum constant declaration has the typical format "`Name [(Expression {, Expression})]`". We compute the string similarities of (1) names and (2) (optionally) expression lists, averaging them with the context similarity of enum constants.

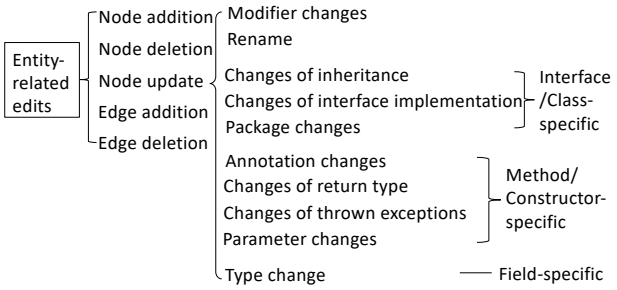

Figure 5: The types of edits that BUCOND recognizes

At the end of this step, BUCOND updates M by adding all nodes matched based on similarities, and removing those nodes from the unmatched sets. For the PEGs of Figure 4, this step detects the mapping of `C.m(...)` between r and b, because the two methods have the same name `m`, same context, and similar bodies.

3.2.3 Edit Generation. Based on the mapping results, this step (1) recognizes five major categories of entity-related edits: node addition, node deletion, node update, edge addition, and edge deletion (see Figure 5), and (2) links branches.

Edit Identification. For each unmatched node between graphs, BUCOND infers the *entity addition or deletion* by either branch. For instance, between G_l and G_b, BUCOND considers an unmatched node in G_l to imply an entity addition, and derives an entity deletion from any unmatched node in G_b. It records add/delete operations in the edited nodes to facilitate later data queries. For matched nodes, BUCOND compares the code details to generate *entity updates* as needed. For instance, if two matched nodes have distinct names, BUCOND generates a rename operation and stores that edit inside the **branch node** (i.e., the edited node in either G_l or G_r). Similarly, if two matched nodes have distinct modifiers (e.g., `public` vs. `private`), BUCOND creates a modifier update. If two matched nodes have distinct incoming edges, BUCOND compares edges and their types, to record *edge additions* and *edge deletions* inside the branch node. We denote all identified edits by l and r with Δ_l and Δ_r.

Branch Linking. Both Δ_l and Δ_r are described with respect to the common base b. However, if we simply use these edits to infer potential build conflicts, we have to frequently map edited nodes in one branch (e.g., l) to b, and map those edits to the other branch (e.g., r) for conflict reasoning. To avoid redirecting the mapping via b, BUCOND links nodes between G_l and G_r based on their separate mappings with G_b. Specifically, if an updated or unchanged node $n_l \in G_l$ is mapped to $n_b \in G_b$ which is further mapped to $n_r \in G_r$, then BUCOND adds a direct link between n_l and n_r. For any node n deleted by either branch, BUCOND copies n from the base version to that branch's graph and marks the copy as "deleted". In this way, BUCOND ensures that every node in G_b can find a counterpart in the other two graphs and adds direct links between G_l and G_r. Once all links are established, BUCOND can freely switch between the branch graphs without revisiting G_b anymore. We denote the linked revised graphs with G'_l and G'_r.

3.3 Phase III: Pattern Matching

Our research novelty mainly lies in this phase. We defined a pattern set to comprehensively enumerate the possible cases where

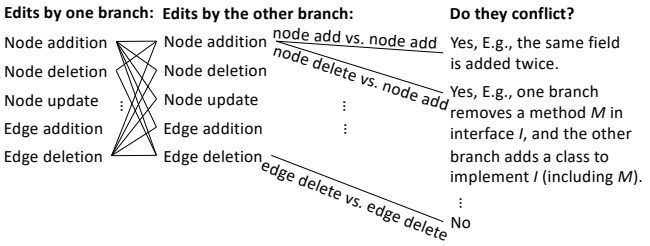

Figure 6: Our exploration procedure of conflict patterns

the combination of branch edits can trigger build conflicts (Section 3.3.1). Accordingly, BUCOND performs pattern matching on the edits embedded in G'_l and G'_r to detect conflicts (Section 3.3.2).

3.3.1 Pattern Definition. Our preliminary study in Section 2 shows that when cross-branch edit combinations violate the def-use constraints between entities, build conflicts occur. Thus, we systematically explored all possible cross-branch combinations between the five major edit types shown in Figure 5, assessed whether a build error can occur for each combination, and defined conflict patterns for all recognized combinations that can trigger build errors.

Figure 6 visualizes our exploration process. We enumerated edit combinations between branches to decide whether any combination can trigger build errors. To recognize the conflicting scenarios for every combination, we considered (1) all possible node/entity types, (2) all possible ways to use an entity, (3) all possible update operations applicable to any entity type (e.g., method renaming or parameter-list changes), and (4) whether the co-applied edits involve def/use of the same entity or distinct entities in the same class hierarchy. For each enumerated scenario, we assessed whether a build conflict can occur based on the four def-use constraints mentioned in Section 2. Namely, if a scenario violates any def-use constraint, the edit combination triggers a build error, and thus the combined edits *conflict* with each other. Notice that we do not explore the scenarios where combined edits trigger both textual and build conflicts (e.g., both branches insert defs of the same method at the same location). As current tools can detect textual conflicts, our research focuses on the scenarios overlooked by prior work.

Since there is no prior knowledge of all possible conflicting scenarios, we spent lots of time to enumerate edit combinations and to identify conflicting scenarios. In particular, the first author did the systematic exploration mentioned above to develop an initial pattern set. Afterwards, all authors held multiple meetings to discuss and iteratively improve the pattern set. We regularly searched for real build conflicts in open-source projects to check whether our pattern set covers them; if not, we added those missing patterns to ensure comprehensiveness. In total, we spent six months defining and refining conflict patterns.

Table 4 summarizes the 57 conflict patterns. As shown in the table, the patterns are derived from six cross-branch edit combinations. Most of these patterns (i.e., 30) describe the conflicting scenarios where one branch updates the def of an entity (e.g., a class or a method), and the other branch adds relevant entity uses. The 30 patterns are different in terms of the node types, finer-grained update categories (e.g., modifier changes vs. rename), and edge types. Another 11 patterns are about the scenarios where one branch updates an entity, and the other branch adds an entity related to the old version of updated entity. Nine patterns are about

Table 4: The 57 conflict patterns we identified

Edit Combination	# of Patterns	Exemplar Pattern	Description of The Exemplar Conflict Pattern
Node update vs. Edge addition	30	Field: change modifier to `final` vs. add write access	One branch makes a field `final`, while the other branch adds code to write a value to the field.
Node update vs. Node addition	11	Class: change a class to an abstract one vs. add sub class	One branch revises a concrete class to be an abstract one, and adds abstract method declarations; the other branch creates a new class to extend the original class, without overriding the abstract methods or declaring itself to be abstract.
Node deletion vs. Edge addition	9	Method: remove def vs. add use	One branch removes the definition of a Java method, while the other branch adds invocation(s) of that method.
Node addition vs. Node addition	4	Field: add def vs. add def	One branch adds a field in class `A`, while the other branch inserts the same field at a different program location in `A`.
Node addition vs. Edge addition	2	Class: add method def in super vs. change a class to extend the super class	One branch adds a method `foo()` in a class `A`; the other branch updates class `B` to extend `A`. `B` has an existing definition of `foo()`, whose return type or modifiers are inconsistent with the super method.
Node deletion vs. Node addition	1	Interface: remove method def vs. add class to implement the super	One branch removes method `foo()` from an interface `I`; the other branch creates a class to implement `I` and annotates its implementation for `foo()` with `@Override`.

Table 5: Utility functions defined to query graphs G'_l and G'_r

Category	Functions
Common	`getName()`, `getNewRefs(...)`, `getParent()`, `hasModifierChanged(...)`, `isAbstract()`, `isAdded()`, `isRenamed()`, `isRemoved()`, `otherBranch()`.
Function-specific	`getNewExceptions()`, `getOverridingMethods()`, `hasExceptionChanged()`, `hasParamChanged()`, `hasReturnChanged()`, `isOverridden()`
Type-specific	`getFields()`, `getImports()`, `getMethods()`, `hasPackageChanged()`, `isNewlyExtended()`, `isNewlyImplemented()`
Field-specific	`hasTypeChanged()`

the scenarios where one branch deletes an entity, and the other branch adds reference(s) to the original entity. The remaining seven patterns correspond to another three edit combinations. Due to the space limit, Table 4 only shows a subset of all patterns in Bucond. These exemplar patterns do not overlap with the ones shown in Table 2, although both sets are covered by Bucond. Please refer to our open-sourced dataset for more details of the patterns.

3.3.2 Matcher Implementation. We first defined reusable utility functions to query graphs for various edits on nodes or edges. Based on those utility functions, we created a set of matchers to match the edits embedded in G'_l and G'_r with known patterns. As shown in Table 5, there are four kinds of utility functions: common, function-specific, type-specific, and field-specific. **Common** functions can be invoked on almost all entity types. **Function-specific** ones can be invoked on two specialized entity types: methods and constructors. **Type-specific** functions are callable on two entity types: classes and interfaces. **Field-specific** means that the function `hasTypeChanged()` is only invokable on field entities.

The utility functions either (1) query attributes of any given entity, (2) check for any entity's editing status, or (3) retrieve edit details. For instance, if `getFields()` is called on a class, the return value is a list of fields defined by that class. If `hasParamChanged()` is called on a method, a boolean value is returned to imply whether the method's parameter list is updated. If `getNewRefs(...)` is called on an entity e, the return value is a list of entities that have newly add edges pointing to e. When `otherBranch()` is called on an entity in one graph (e.g., G'_l), the entity's counterpart in the other graph (e.g., G'_r) is returned for further comparison.

We successfully implemented 57 matchers for the identified patterns using the above utility functions. Algorithm 2 presents the pseudocode of one exemplar matcher in Bucond. For instance, to identify all conflicts of type "Method: remove def vs. add use", the matcher enumerates all method nodes in both graphs. For each enumerated node m, the matcher checks whether the node is labeled "deleted" while its counterpart in the other graph remains unchanged. If so, the matcher further checks whether the counterpart has any *use*-typed edge (e.g., "calls") added. A conflict is reported whenever m satisfies all the above conditions.

Algorithm 2: The matcher that identifies conflicts of type "Method: remove def vs. add use"

```
2.1 conflicts ← ∅;
2.2 foreach m in allMethods do
2.3     if m.isRemoved() && m.otherBranch().noChange then
2.4         if m.otherBranch().getNewRefs() != ∅ then
2.5             // report conflict details
```

4 EVALUATION

To assess the effectiveness of Bucond, we explored the following three research questions (RQs):

- **RQ1:** Can Bucond identify various conflicts correctly?
- **RQ2:** How effective Bucond is in identifying real-world build conflicts?
- **RQ3:** How effectively does Bucond work when the automatic build is inapplicable to detect conflicts?

The following subsections will describe the datasets, evaluation metrics, and experiment results. Our evaluation was conducted on a computer with Intel (R) Core (TM) i5-4210U CPU @2.40GHz, 8 GB memory, and Windows 8 OS.

4.1 Datasets

There is no publicly available dataset of build conflicts in Java programs, so we created three datasets for tool evaluation.

4.1.1 ***Dataset 1****.* This dataset contains 57 merging scenarios we manually crafted, to assess the implementation status of Bucond's pattern-matching logic (RQ1). Each scenario has exactly one conflict, corresponding to one of the patterns Bucond handles. We prepared three program versions for each scenario: b, l, and r, and recorded the conflict detail as ground truth.

4.1.2 ***Dataset 2****.* This dataset contains 55 real merging scenarios, among which 81 conflicts triggered build errors. We used this dataset to assess how well Bucond performs when identifying real build conflicts (RQ2). We found these conflicts and labeled them in the following way. First, we ranked Java projects on GitHub based

Table 6: Distribution of the 81 conflicts in *Dataset 2*

Conflict Type	# of Conflicts
Class: add method def in super vs. add sub class	2
Class: change a method's parameter list in super vs. add sub class	2
Class: change a method's return type in super vs. add sub class	1
Class: remove def vs. add use	9
Class: rename def vs. add use	7
Constructor: change the parameter list vs. add use	5
Field: add def vs. add def	3
Field: change a field's type vs. add use	1
Field: remove def vs. add use	4
Import: remove def vs. add use	7
Interface: add method def in super vs. add class to implement the super	6
Interface: change a class to implement the super vs change a method's return type in the class	9
Interface: change a method's parameter list in super vs. add class to implement the super	2
Interface: remove method def in super vs. add class to implement the super	1
Interface: rename a method in super vs. add class to implement the super	1
Local Variable: move def into an `if`-block vs. split that `if`-block into two	2
Method: change the parameter list vs. add use	3
Method: change the return type vs. add use	1
Method: remove def vs. add use	8
Method: rename def vs. add use	5
Package: rename def vs. add use	2

on their popularity (i.e., star counts), and then cloned repositories for the top 1,000 projects. Next, we only kept the projects that can be built with Maven [7], Ant [26], or Gradle [4], as we relied on these build tools to compile each naïvely merged version A_m. Afterwards, we removed tutorial projects as they are not real Java applications and may not show real-world merging scenarios. Starting with the refined 209 repositories, we identified 117,218 merging scenarios by searching for any commit with two parent/predecessor commits.

We processed each merging scenario in three steps to create the ground truth of real build conflicts. *Step 1* applies git-merge to l and r to generate a text-based merged version A_m. If A_m contains any textual conflict, we discard the scenario. Otherwise, if A_m has zero textual conflict, in *Step 2*, we try to build l, r, and A_m. If both l and r build successfully but A_m does not, we conclude that the scenario has at least one build conflict. In *Step 3*, for each revealed build error in A_m, we use the error as guidance, analyze program differences among versions (b, l, r, A_m, m), look for edited code responsible for that error, and label the scenario if we find conflicting branch edits in Java code as the root cause. In this procedure, we found 15,886 scenarios to have textual conflicts and 55 scenarios to have 81 build conflicts. Table 6 shows the distribution of 81 build conflicts based on their types. As shown in the table, the build conflicts are diverse, belonging to 21 types, 20 of which are in our 57-pattern set.

4.1.3 ***Dataset 3.*** This dataset has 13 real merging scenarios, with 17 conflicts found via manual inspection. We used this dataset to assess how effectively BUCOND works when merging scenarios have both textual and build conflicts (RQ3). Notice that in such scenarios, the A_m produced by git-merge has textual conflicts, so automatic build is inapplicable. Developers must manually resolve all textual conflicts before using automatic build to find build conflicts. We envision BUCOND to make up for the limitations of git-merge and automatic build. In other words, the application of both BUCOND and git-merge can give developers a global overview of the co-existence between textual and build conflicts, before developers attempt to resolve any conflict. The global view can give developers more comprehensive information, faciltating them to make better decisions on how to resolve individual conflicts.

Table 7: Distribution of the 17 conflicts in *Dataset 3*

Conflict Type	# of Conflicts
Class: rename def vs. add use	2
Constructor: change the parameter list vs. add use	2
Field: remove def vs. add use	1
Import: remove def vs. add use	4
Interface: change a method's return type in super vs. add class to implement the super	2
Interface: rename def vs. add use	1
Local Variable: add def vs. add def	1
Method: remove def vs. add use	1
Method: rename vs. add use	3

To find conflicts manually, we randomly picked nine popular open-source Java repositories: Activiti [10], pebble [19], fastjson [13], vectorz [23], nuxeo [18], wildfly [25], webmagic [24], truth [22], and elasticsearch [12]. We filtered out the scenarios where no textual conflict is reported by git-merge. Among the remaining scenarios, we manually compared all versions involved: l, r, b, m, and A_m. Based on our understanding of program context and the semantics of branch edits, we speculated the semantics of naïve edit combination across branches, analyzed the potential build errors that can be triggered, and identified conflicting edits. For instance, in a merging scenario of fastjson [1], we observed that l adds a reference to an imported class `GenericArrayType`, and r removes that class import from the same file. Thus, we speculated the naïve integration of branch edits to cause a broken def-use link for `GenericArrayType`. Our further examination of m confirmed the speculation, as developers added back the removed class import in m. In this way, we found a build conflict. Table 7 shows all conflicts we manually identified.

Among the 3 datasets, Dataset 1 covers the most conflict types (i.e., 57), as we crafted the synthetic conflicts to expose BUCOND to diverse merging scenarios. Dataset 2 contains the most conflicts (i.e., 81). Dataset 3 has the fewest conflicts (i.e., 17) and covers only 9 types, because manually detecting conflicts is very time-consuming. To ensure that we collected conflicts without bias towards our tool, we had one author independently mine software repositories for Datasets 2&3, and had another author separately create our tool. Once the datasets were created, three authors inspected all included conflicts to ensure the correctness of ground truth.

4.2 Metrics

The following metrics are used to evaluate conflict detectors.

Precision (P) measures among all reports generated by a detector, how many of them are true positives:

$$P = \frac{\text{\# of correct reports}}{\text{Total \# of reported conflicts}}$$

For Dataset 1, we used the labeled data to calculate precision automatically. Suppose that we have a set of labeled conflicts S_1, and the set of reported conflicts is S_2. We use $|S_1 \cap S_2|/|S_2|$ to compute precision. The labeled ground truth in Datasets 2 & 3 can be incomplete, due to the limitation of compiler-based detection and manual detection. Thus, in addition to $|S_1 \cap S_2|$, we also manually checked the reported conflicts not covered by ground truth, to reveal additional

Table 8: The merging scenario where BUCOND missed two build conflicts [8]

Changes in l	Changes in r
- if((api != null)...) {	if ((api != null)...){
+ final boolean readable = (api != null) ... ;	
+ if(readable) {	
...	...
String produces = ... ;	String produces = ... ;
String consumes = ... ;	String consumes = ... ;
... // code_block_1	... // code_block_1
+ }	
+ if (api == null || readable) {	
... // code_block_2	... // code_block_2
	+ String[] apiConsumes = consumes;
	+ String[] apiProduces = produces;

correct reports and compute precision accordingly. For each report not matching the ground truth, our manual checking compares b, l, r, and m. We consider a reported conflict to be correct if (1) the branch edits are not naïvely integrated into m, (2) m modifies part of the branch edits, and (3) the modification can fix any build error triggered by a naïve integration of branch edits.

Recall (R) measures among all known true positives, how many of them are reported by a detector:

$$R = \frac{\text{\# of retrieved conflicts}}{\text{Total \# of known conflicts}}$$

We relied on the labeled data in all datasets to compute recall. Suppose that the labeled conflict set is S_1, and the reported conflict set is S_2. We use $|S_1 \cap S_2|/|S_1|$ to compute recall.

F score (F) is the harmonic mean between precision and recall. It reflects the trade-off between those two metrics.

$$F = \frac{2 \times P \times R}{P + R}.$$

All metrics have values within [0, 1]: the higher value, the better.

4.3 Experiment Results on Dataset 1

We applied BUCOND to all the synthetic merging scenarios in Dataset 1, and checked if it detects all labeled conflicts correctly. The measured precision, recall, and F-score rates are all 100%. By covering all 57 patterns, this dataset enabled us to assess BUCOND's capability of handling distinct build conflicts.

Finding 1: *BUCOND identified 57 types of conflicts correctly, showing great capability of handling diverse conflicting scenarios.*

4.4 Experiment Results on Dataset 2

When applied to Dataset 2, BUCOND reported 79 conflicts, 77 of which exist in ground truth. Our manual inspection shows that the remaining two conflicts are also real (i.e., true positives). BUCOND missed four known conflicts in the labeled dataset. Therefore, BUCOND achieved 100% precision (79/79), 95% recall (77/81), and 97% F-score. In one scenario, BUCOND was able to identify two more conflicts than compiler-based conflict detection. Thanks to its usage of static analysis, BUCOND did not get stuck with the compilation errors resulting from initially found conflicts.

We manually inspected the four false negatives—the conflicts missed by BUCOND, and found two reasons. First, conflicts were concerning the def and use of local variables. A program usually has a lot more local variables (LVs) than entities. Thus, BUCOND does not model LVs in graphs for efficiency, nor does it detect LV-related conflicts. As shown in Table 8, a merging scenario has two conflicts separately related to variables `produces` and `consumes`. The conflicts happened because l split an `if`-statement into two: the first `if`-statement defines both variables; the second one contains `code_block_2`, which does not use any of the variables. Meanwhile, r inserted usage of both variables at the end of `code_block_2`. The naïve integration of branch edits caused the newly added variable usage to be out of the scope of variable definitions.

Table 9: The merging scenario where BUCOND missed a build conflict related to an `import`-declaration [9]

Changes in l	Changes in r
- import java.net.*;	import java.net.*;
...	...
	+ } catch (SocketTimeoutException ste) {
	+ throw ste;

The other two false negatives were related to `import`-declarations. As shown in Table 9, while l removes the `import`-declaration for classes `java.net.*`, r adds a ref to the class `java.net.SocketTimeoutException`. Because there is no explicit mapping between the removed classes represented by wildcard "`*`" and the added class usage, BUCOND could not identify the build conflict. In the future, we plan to improve BUCOND to analyze software libraries, better interpret the meaning of wildcards, and recognize such conflicts.

Finding 2: *On Dataset 2, BUCOND detected conflicts with 100% precision, 95% recall, and 97% F-score. It means that BUCOND can identify build conflicts with high precision and high recall.*

As a software merge tool, IntelliMerge [56] creates program element graphs for b, l, and r. It compares graphs to detect refactoring operations, which help improve the results of element matching and software merge. IntelliMerge seems to be able to handle build conflicts when the conflicting edits are relevant to refactorings (e.g., entity renaming or removal). Because BUCOND has a similar approach design to IntelliMerge in terms of graph construction and graph comparison, we were curious how BUCOND compares with IntelliMerge when detecting build conflicts. Therefore, we also applied IntelliMerge to Dataset 2. Our experiment shows that IntelliMerge only detected and resolved four build conflicts, all of which were of the type *Import: remove def vs. add use*. Our observation means that IntelliMerge rarely detects build conflicts. BUCOND outperforms IntelliMerge by identifying a lot more build conflicts.

Discussion. BUCOND is different from IntelliMerge in terms of the research objective, approach design, and implementation. IntelliMerge aims at detecting *textual conflicts* more accurately than text-based merge, while BUCOND intends to detect *build conflicts* via static analysis, without using automatic build. In terms of approach design, IntelliMerge textually compares the edits that are simultaneously applied by distinct branches to the *same* or *aligned* entities. However, BUCOND compares the edits simultaneously applied by distinct branches to *different* but *related* entities. Such an edit comparison is far more complicated than text-based comparison, as it requires for extensive semantic reasoning. Thus, BUCOND novelly defines a set of 57 patterns to represent the scenarios where co-applied edits can introduce build conflicts. It also defines 57 novel pattern-matchers to reason about the semantics of edits.

In terms of implementation, as IntelliMerge focuses on entity alignment, it does not carefully model or align edges as what BUCOND does. For instance, IntelliMerge provides insufficient or no support for modeling (1) `calls/initializes`-edges introduced by `this`-expressions (e.g., `this(...)`) and field declarations (e.g., `A a = B.foo(...)`), (2) `imports`-edges pointing to the entities defined by JDK or third-party libraries (e.g., `import java.util.List`), (3) `reads`-edges pointing to enum constants. BUCOND models all these edges.

As mentioned in Section 1, **compiler-based tools** (i.e., Crystal and WeCode) detect build conflicts via compilation instead of static analysis, so they report only build errors instead of the responsible conflicting edits. Additionally, both tools are unavailable, so we were unable to run either tool for the empirical comparison with BUCOND. However, our experiment with Dataset 2 can still simulate an indirect comparison between compiler-based tools and BUCOND. Specifically, because all 81 conflicts were manually located based on the build errors in A_m, theoretically speaking, compiler-based tools can report those build errors as hints of the 81 conflicts. As shown by our experiment results, BUCOND independently reported 79 conflicts, with 2 of the conflicts not implied by any build error. Our observations indicate that BUCOND complements compiler-based tools in two ways: (1) it pinpoints build conflicts; (2) it can reveal conflicts not implied by any observed build errors.

Finding 3: *On Dataset 2, BUCOND detected a lot more build conflicts than IntelliMerge (79 vs. 4). BUCOND complements IntelliMerge by offering a better support for build-conflict detection.*

4.5 Experiment Results on Dataset 3

For Dataset 3, BUCOND reported 19 conflicts, 15 of which match the ground truth. Our manual inspection shows that the remaining four conflicts are also true positives. BUCOND missed two known conflicts. The first conflict was originally introduced by duplicated additions of the same local variable; the second one was similar to the conflict shown in Table 9. Because BUCOND does not track local variables or interpret wildcards used in `import`-declarations, it could not recognize the conflicts. In summary, BUCOND achieved 100% precision (19/19), 88% recall (15/17), and 94% F-score.

Table 10 shows the time cost of BUCOND when it was applied to Dataset 3. The three columns under **# of Analyzed Files** separately count the edit-relevant Java files in b, l, and r. The columns under **# of Analyzed Entities** count the total number of entities included in those files. BUCOND's time cost often increases with the number of analyzed files or entities because given a scenario, BUCOND spent over 99% of execution time on PEG construction and comparison. As there are more entities and more inter-entity relations, the graphs can become more complex, PEG comparison can become more time-consuming and thus BUCOND's runtime overhead grows.

This experiment simulates another indirect comparison between BUCOND and compiler-based tools. Specifically in Dataset 3, because textual conflicts coexist with build conflicts in each merging scenario, none of the automatically merged versions A_m is compilable. Automatic build is inapplicable and compiler-based tools cannot report build errors for any of the known 17 conflicts. Our results show that BUCOND independently detected 15 of the 17 conflicts, and found 4 extra conflicts not covered by the ground truth. These observations imply that BUCOND complements compiler-based tools.

Table 10: The time cost of BUCOND when it was applied to the 13 merging scenarios in Dataset 3

Idx	# of Analyzed Files			# of Analyzed Entities			Time Cost (minute)
	b	l	r	b	l	r	
1	196	198	207	4,817	4,840	4,933	11.8
2	43	55	49	2,462	2,694	2,503	2.8
3	158	167	158	5,804	5,952	5,797	21.6
4	137	141	137	2,391	2,520	2,396	3.2
5	10	11	12	1,281	1,323	1,302	1.5
6	21	20	22	631	624	639	0.3
7	83	91	86	1,596	1,770	1,640	1.4
8	9	13	27	214	315	524	0.1
9	49	58	51	1,683	1,790	1,713	0.9
10	9	10	9	951	971	959	0.4
11	11	12	13	150	161	164	0.1
12	137	145	132	4,345	4,516	4,220	4.6
13	125	137	125	4,410	4,746	4,423	7.5

Finding 4: *On Dataset 3, BUCOND detected conflicts with 100% precision, 88% recall, and 94% F-score. It complements compiler-based tools (e.g., Crystal) in two ways: reporting build conflicts instead of build errors and bypassing automatic build.*

5 THREATS TO VALIDITY

Threats to External Validity. The evaluation is based on 155 labeled conflicts, so our observations may not generalize well to conflicts outside the evaluation datasets. We have spent one year collecting data of build conflicts, so the current datasets are the best options we have for tool evaluation now. The major difficulty of creating large-scale datasets is that compiler-based conflict detection has great limitations when being applied to open-source repositories: they do not work when branches-to-merge have textual conflicts and most conflicting merging scenarios have textual conflicts (see Section 4.1.2). In the future, we plan to expand the evaluation datasets to make our findings more representative.

Threats to Construct Validity. We defined 57 conflict types based on (1) the observations of real conflicts and (2) the generalization of observations. BUCOND shares the same limitation with existing static analysis-based tools, for being sound but incomplete. However, as a complementary tool to compiler-based tools, BUCOND can help developers better understand the merging scenarios when both build and textual conflicts exist. Its high detection precision implies that the tool can always report conflicts reliably. According to our experience, the 57 types cover all observed conflicts except for those related to (1) local variables or (2) wildcard usage.

Threats to Internal Validity. BUCOND uses JavaSymbolSolver to resolve identifier bindings. When JavaSymbolSolver fails, BUCOND implements a naïve approach that applies string matching to function names and parameter lists, in order to infer the caller-callee relations with best effort. Similarly, BUCOND also applies string matching to resolve type bindings for field accesses. Although this approach does not guarantee to always successfully resolve bindings, it has worked well so far.

When matching nodes between graphs, BUCOND reuses the parameter settings of existing work [56] to decide whether two nodes are similar enough to match. These settings include the similarity threshold 0.618 and the 3-grams used for string partition. We did not explore different value settings to find the best configuration. Intuitively, as the values increase, it becomes harder for BUCOND to find matches between nodes, while the matched nodes are often

very similar to each other. Meanwhile, as the parameter values decrease, it becomes easier for BUCOND to match nodes, although the quality of matches may suffer. As prior work shows that both parameter settings lead to reasonably good results, we reused the settings and also observed them to work well in BUCOND.

6 RELATED WORK

Our research is related to automated software merge, awareness-raising tools, and empirical studies on merge conflicts.

6.1 Automated Software Merge

Tools were built to detect or resolve merge conflicts [6, 28, 29, 33–35, 37, 40, 44, 48, 56, 57, 62]. For instance, FSTMerge [6, 29, 35] parses code for ASTs, and matches nodes between l and r purely based on the class or method signatures; it then integrates the edits inside each pair of matched methods via textual merge. JDime [28] also matches Java methods and classes based on syntax trees. However, unlike FSTMerge, JDime merges edits inside methods based on ASTs. It can report conflicts more precisely than FSTMerge [36]. AutoMerge [62] also detects conflicts based on AST comparison. However, going beyond conflict detection, AutoMerge attempts to resolve conflicts by proposing alternative strategies to merge l and r, with each strategy integrating branch edits in a distinct way. DeepMerge [37] uses deep learning to resolve textual conflicts. None of the tools mentioned above detect higher-order conflicts.

SafeMerge [57] takes in b, l, r, and m, for a given merging sceanrio. It statically infers the relational postconditions of distinct versions to model program semantics. By comparing postconditions, SafeMerge decides whether m is *free of conflicts*, i.e., without introducing new semantics nonexistent in l or r. SafeMerge cannot effectively detect build conflicts, as it does not relate edits applied to distinct entities for semantic reasoning. MrgBldBrkFixer [58] compares the ASTs of C++ files. It detects and resolves the build conflicts related to (1) renamed entities (e.g., class renaming), and (2) changes to the parameter/return types of functions. Wuensche et al. also created a build-conflict detector for C++ code [59]. The tool statically analyzes call graphs to reveal three causes for conflicts: (1) changes to method signatures (i.e., modified names/arguments/return values) and complete entity removals, (2) missing `#include`-statements, and (3) duplicate definitions of functions or variables.

Our pattern set is more comprehensive than the conflict types considered by prior build-conflict detectors. As with prior work, BUCOND models the *calls*, *imports*, *declares*, and *contains* relations between entities to capture conflicts related to (1) renamed or removed entities, (2) duplicated entities, (3) signature changes of functions, and (4) import declarations. However, different from prior work, BUCOND also models five other types of inter-entity relations (see Section 3) to capture conflicts related to edited class inheritance, interface implementation, class initialization, and field access.

6.2 Awareness-Raising Tools

Several tools [30, 33, 34, 40, 42, 43, 46, 54] were created to monitor and compare programmers' development activities, and to improve team activity awareness. For instance, Palantír [54] informs a developer of the artifacts changed by other developers, calculates the severity of those changes, and visualizes the information. Cassandra [42] is a conflict minimization technique. It observes the super-sub and caller-callee dependencies between program entities. By treating those dependencies as constraints on file-editing tasks, Cassandra identifies tasks that will conflict when performed in parallel. It then schedules tasks to recommend conflict-free development paths. None of these tools localize merge conflicts.

6.3 Empirical Studies on Merge Conflicts

Some studies were conducted to characterize the relationship between merge conflicts and developers' coding activities [27, 38, 45, 47, 50, 52]. For instance, Leßenich et al. surveyed 41 developers and identified 7 potential indicators (e.g., # of changed files in both branches) for merge conflicts [45]. Mahmoudi et al. observed that certain refactoring types (e.g., Extract Method) are more related to conflicts [47]. Other studies characterize the root causes or resolutions of conflicts [2, 32, 51, 53, 55, 60]. Specifically, Shen et al. [55] manually inspected three types of conflicts: textual, build, and test conflicts. They reported that higher-order conflicts are hard to detect and resolve, although existing tools mainly focus on textual conflicts. Inspired by the study by Shen et al., we developed BUCOND to reduce the technical barrier of detecting build conflicts.

7 CONCLUSION

Software merge is complex and time-consuming. Although several tools can detect textual conflicts, we found few tools to detect build conflicts. Our preliminary study with build conflicts reveals the typical constraints that conflict-free software merge should satisfy. Such observations motivated us to create BUCOND. Our evaluation with three datasets shows exciting results. BUCOND detected conflicts with high precision and high recall. Although it missed some conflicts detected by automatic build or manual inspection, it managed to reveal more conflicts when (1) textual and build conflicts coexist, or (2) compiler-based conflict detection is stuck with the build errors triggered by initially revealed conflicts. BUCOND complements existing tools due to its usage of static analysis and the comprehensive pattern set of conflicts.

We made three major contributions. First, we defined a novel pattern set to enumerate 57 types of conflict-triggering edit combination. This set is based on our preliminary study, the systematic exploration of edit combinations, and frequent crawling for real build conflicts. The process is very challenging, demanding significant creativity and brainstorming among authors. We spent six months defining and refining those patterns. Second, BUCOND is the first static analysis-based tool that detects Java build conflicts effectively. Third, we evaluated BUCOND using three datasets. All conflicts in Datasets 2&3 are from open-source repositories. We spent one year creating the datasets. No prior work provides such comprehensive datasets of real build conflicts.

The approach design (including the 57 patterns) of BUCOND can be reimplemented for different object-oriented programming languages (e.g., C#) to detect conflicts in non-Java projects. In the future, we will extend BUCOND to also resolve build conflicts.

ACKNOWLEDGMENTS

This work was supported by NSF-1845446 and NSF-2106420. We thank reviewers for their valuable feedback.

REFERENCES

[1] 2017. Merge branch 'master' into master. https://github.com/alibaba/fastjson/commit/b9d301d6.

[2] 2020. On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub. *IEEE Transactions on Software Engineering* 46, 8 (2020), 892–915. https://doi.org/10.1109/TSE.2018.2871083

[3] 2021. Golden Ratio. https://mathworld.wolfram.com/GoldenRatio.html.

[4] 2021. Gradle. https://gradle.org.

[5] 2021. JavaParser. https://javaparser.org.

[6] 2021. jFSTMerge. https://github.com/guilhermejccavalcanti/jFSTMerge.

[7] 2021. Maven. https://maven.apache.org.

[8] 2021. Merge branch 'jaxrs_reader' of ssh://github.com/lugaru1234/swagger-core into lugaru1234-jaxrs_reader. https://github.com/swagger-api/swagger-core/commit/95fa219a842325f8d7d5176ec534ceb3e4e5cd9a.

[9] 2021. Merge pull request #81 from Worxfr/master. https://github.com/NanoHttpd/nanohttpd/commit/f81ed131ef9b10e1940f9fd0ed3129e47a4e7b85.

[10] 2022. Activiti. https://github.com/Activiti/Activiti.

[11] 2022. druid. https://github.com/alibaba/druid.

[12] 2022. elasticsearch. https://github.com/elastic/elasticsearch.

[13] 2022. fastjson. https://github.com/alibaba/fastjson.

[14] 2022. JavaPoet. https://github.com/square/javapoet.

[15] 2022. Jedis. https://github.com/redis/jedis.

[16] 2022. litemall. https://github.com/linlinjava/litemall.

[17] 2022. MyBatis-Plus. https://github.com/baomidou/mybatis-plus.

[18] 2022. nuxeo. https://github.com/nuxeo/nuxeo.

[19] 2022. pebble. https://github.com/PebbleTemplates/pebble.

[20] 2022. Redisson. https://github.com/redisson/redisson.

[21] 2022. Spring Cloud Alibaba. https://github.com/alibaba/spring-cloud-alibaba.

[22] 2022. truth. https://github.com/google/truth.

[23] 2022. vectorz. https://github.com/mikera/vectorz.

[24] 2022. webmagic. https://github.com/code4craft/webmagic.

[25] 2022. wildfly. https://github.com/wildfly/wildfly.

[26] Last visited 07/18/19. Ant. https://ant.apache.org.

[27] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma. 2017. An Empirical Examination of the Relationship between Code Smells and Merge Conflicts. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 58–67. https://doi.org/10.1109/ESEM.2017.12

[28] Sven Apel, Olaf Lessenich, and Christian Lengauer. 2012. Structured Merge with Auto-tuning: Balancing Precision and Performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. ACM, New York, NY, USA, 120–129. https://doi.org/10.1145/2351676.2351694

[29] Sven Apel, Jorg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kastner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/2025113.2025141

[30] Jacob T Biehl, Mary Czerwinski, Greg Smith, and George G Robertson. 2007. FASTDash: a visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1313–1322.

[31] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. 2014. Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/2635868.2635880

[32] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. 2020. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering* 25, 1 (2020), 562–590. https://doi.org/10.1007/s10664-019-09735-4

[33] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 168–178. https://doi.org/10.1145/2025113.2025139

[34] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *IEEE Transactions on Software Engineering* 39, 10 (Oct 2013), 1358–1375. https://doi.org/10.1109/TSE.2013.28

[35] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 59 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133883

[36] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. 2019. The Impact of Structure on Software Merging: Semistructured versus Structured Merge. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1002–1013. https://doi.org/10.1109/ASE.2019.00097

[37] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu Lahiri. 2022. DeepMerge: Learning to Merge Programs. *IEEE Transactions on Software Engineering* (2022), 1–16. https://doi.org/10.1109/TSE.2022.3183955

[38] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer. 2014. Awareness and Merge Conflicts in Distributed Software Development. In *2014 IEEE 9th International Conference on Global Software Engineering*. 26–35. https://doi.org/10.1109/ICGSE.2014.17

[39] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[40] Mário Luís Guimarães and António Rito Silva. 2012. Improving Early Detection of Software Merge Conflicts. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 342–352. http://dl.acm.org/citation.cfm?id=2337223.2337264

[41] Zijian Jiang, Hao Zhong, and Na Meng. 2021. Investigating and recommending co-changed entities for JavaScript programs. *Journal of Systems and Software* 180 (2021), 111027. https://doi.org/10.1016/j.jss.2021.111027

[42] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 732–741.

[43] Michele Lanza, Marco D'Ambros, Alberto Bacchelli, Lile Hattori, and Francesco Rigotti. 2013. Manhattan: Supporting real-time visual team activity awareness. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 207–210.

[44] Olaf Leßenich, Sven Apel, and Christian Lengauer. 2014. Balancing precision and performance in structured merge. *Automated Software Engineering* 22 (2014), 367–397.

[45] Olaf Leßenich, Janet Siegmund, Sven Apel, Christian K´astner, and Claus Hunsen. 2018. Indicators for Merge Conflicts in the Wild: Survey and Empirical Study. *Automated Software Engg.* 25, 2 (June 2018), 279–313. https://doi.org/10.1007/s10515-017-0227-0

[46] Chandra Maddila, Nachiappan Nagappan, Christian Bird, Georgios Gousios, and Arie van Deursen. 2021. ConE: A Concurrent Edit Detection Tool for Large ScaleSoftware Development. *arXiv preprint arXiv:2101.06542* (2021).

[47] M. Mahmoudi, S. Nadi, and N. Tsantalis. 2019. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 151–162. https://doi.org/10.1109/SANER.2019.8668012

[48] T. Mens. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 28, 5 (2002), 449–462. https://doi.org/10.1109/TSE.2002.1000449

[49] Dimitrios Michail, Joris Kinable, Barak Naveh, and John V Sichi. 2020. JGraphT—A Java Library for Graph Data Structures and Algorithms. *ACM Transactions on Mathematical Software (TOMS)* 46, 2 (2020), 1–29.

[50] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. 2018. The life-cycle of merge conflicts: processes, barriers, and strategies. *Empirical Software Engineering* (2018), 1–44.

[51] Hoai Le Nguyen and Claudia-Lavinia Ignat. 2018. An Analysis of Merge Conflicts and Resolutions in Git-Based Open Source Projects. *Computer Supported Cooperative Work (CSCW)* 27, 3 (01 Dec 2018), 741–765. https://doi.org/10.1007/s10606-018-9323-3

[52] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. [n.d.]. Predicting Merge Conflicts in Collaborative Software Development. https://arxiv.org/pdf/1907.06274.pdf.

[53] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu Lahiri, and Mike Kaufman. 2021. Can Program Synthesis Be Used to Learn Merge Conflict Resolutions? An Empirical Analysis. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 785–796. https://doi.org/10.1109/ICSE43902.2021.00077

[54] Anita Sarma, Zahra Noroozi, and André Van Der Hoek. 2003. Palantír: raising awareness among configuration management workspaces. In *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 444–454.

[55] Bowen Shen, Muhammad Ali Gulzar, Fei He, and Na Meng. 2022. A Characterization Study of Merge Conflicts in Java Projects. *ACM Trans. Softw. Eng. Methodol.* (jun 2022). https://doi.org/10.1145/3546944 Just Accepted.

[56] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. 2019. IntelliMerge: A Refactoring-Aware Software Merging Technique. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 170 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360596

[57] Marcelo Sousa, Isil Dillig, and Shuvendu Lahiri. 2018. Verified Three-Way Program Merge. In *Object-Oriented Programming, Systems, Languages & Applications Conference (OOPSLA 2018)*. ACM. https://www.microsoft.com/en-us/research/publication/verified-three-way-program-merge/

[58] Chungha Sung, Shuvendu K. Lahiri, Mike Kaufman, Pallavi Choudhury, and Chao Wang. 2020. Towards understanding and fixing upstream merge induced conflicts in divergent forks: an industrial case study. In *ICSE (SEIP)*. ACM, 172–181.

[59] Thorsten Wuensche, Artur Andrzejak, and Sascha Schwedes. 2020. Detecting Higher-Order Merge Conflicts in Large Software Projects. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 353–363. https://doi.org/10.1109/ICST46399.2020.00043

[60] R. Yuzuki, H. Hata, and K. Matsumoto. 2015. How we resolve conflict: an empirical study of method-level conflict resolution. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*. 21–24. https://doi.org/10.1109/SWAN.2015.7070484

[61] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 913–923. https://doi.org/10.1109/ICSE.2015.101

[62] Fengmin Zhu and Fei He. 2018. Conflict Resolution for Structured Merge via Version Space Algebra. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 166 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276536