Automating The Detection and Resolution of Build Conflicts in Software Merge for Java Programs

Sheikh Shadab Towqir

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Application

Na Meng, Chair
Muhammad Ali Gulzar
Eli Tilevich
Francisco Servant
Fei He

December 11, 2025
Blacksburg, Virginia

Keywords: Software Merge, Build Conflict, Static Analysis, Pattern Matching, Program Transformation, Large Language Models (LLMs).

Automating The Detection and Resolution of Build Conflicts in Software Merge for Java Programs

Sheikh Shadab Towqir

(ABSTRACT)

Version control systems (VCS) like Git provide an efficient environment for collaborative software development. However, a major challenge of using such systems is the conflicts that occur when developers try to merge branches. This research focuses on build conflicts– a category of higher-order software merge conflicts. The goal of our research is to develop and implement methodologies that automate the detection and resolution process of build conflicts. Our research consists of three pieces. First, we designed and implemented a graph-based pattern matching approach, BUCOND (Build Conflict Detector), to detect conflicts via static analysis. Our evaluation shows that BUCOND accurately identified build conflicts on both synthetic and real-world datasets; it demonstrated great applicability in scenarios where compiler-based tools are inapplicable. Second, we created a hybrid program transformation approach, BuCoR (Build Conflict Resolver), to opportunistically resolve conflicts. We evaluated BuCoR in real-world merging scenarios, observing its great capabilities of generating syntactically correct resolutions and mimicking human developers' resolution practices. Third, we explored a new approach of resolving conflicts using LLMs, and empirically compared the usage of different LLMs as well as diverse prompt designs. We demonstrate that, when properly guided, LLMs can provide highly accurate conflict resolutions. Our research will help developers detect conflicts more efficiently and resolve conflicts with higher effectiveness as well as rigor. Through addressing issues caused by merge conflicts, it will help improve programmer productivity and software reliability.

Automating The Detection and Resolution of Build Conflicts in Software Merge for Java Programs

Sheikh Shadab Towqir

(GENERAL AUDIENCE ABSTRACT)

Version Control Systems (VCS) like Git are essential for team-based software development but often struggle with merge conflicts. One complex category of such conflicts is build conflicts. This research aims to make it easier to detect and resolve these conflicts automatically. First, BUCOND, a graph-based tool, has been developed to identify build conflicts, even in scenarios where traditional methods may fail. Second, BuCoR, a program modification tool, has been created to provide automated fix suggestions for build conflicts, often matching what a human developer might do. Third, we investigate the use of Large Language Models (LLMs) for the purpose of build conflict resolution. By advancing the understanding and automation of these processes, this research work aims to improve the efficiency and reliability of collaborative software development.

Dedication

Dedicated to my family and friends

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Na Meng, for her guidance and mentorship throughout my doctoral studies. This work would not have been possible without her feedback and encouragement. I am sincerely thankful for her continued direction, which allowed me to reach this point.

I would also like to thank my PhD committee members, Dr. Muhammad Ali Gulzar, Dr. Eli Tilevich, Dr. Francisco Servant, and Dr. Fei He, for their valuable insights and constructive feedback throughout this work. Their perspectives and suggestions played an important role in shaping this dissertation and strengthening my research.

I would like to thank my parents, whose love and sacrifices made it possible for me to pursue this degree. They have been my greatest source of motivation throughout this journey. Thank you, Abbu and Ammu, for everything you have done for me. I would also like to thank my brothers, Nabil and Shakib, whose love, support, and presence have been a constant source of strength for me.

I would like to thank my friends, Mehal, Yasin, Sayeef, Tawsif, and Rakibul, who have been a part of my life for many years. Though our lives have taken different paths, their friendship and support have remained constant. They have stood by me through both my best and most challenging times, and I cannot thank them enough.

My time here would not have been the same without the many friends I made in Blacksburg. I would like to thank: Ranit, Shuchismita, Debasmita, Arit, Arnab, Chiranjib, Poorna, Anish, Priyambada, Prayati, Amartya, Nabayan, Asif, and Arjab. Thank you for welcoming me into your lives, for supporting me as I learned and grew, and for making this place feel

like a home away from home. Your presence has meant a lot to me. I will always cherish the memories we shared.

Finally, I would like to thank everyone who has helped and encouraged me throughout this work. I am grateful for all your support.

Contents

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence

AST Abstract Syntax Tree

BUCOND Build Conflict Detector

BuCoR Build Conflict Resolver

DSL Domain-specific Language

FQN Fully Qualified Name

GPT Generative Pre-trained Transformer

JDK Java Development Kit

LLaMA Large Language Model Meta AI

LLM Large Language Model

ML Machine Learning

PEG Program Entity Graph

VCS Version Control System

Chapter 1

Introduction

In modern software development, collaboration is essential for creating complex and robust software products. Version Control Systems (VCS) like Git [1] play a crucial role in this process, allowing developers to work concurrently on different features or fixes through a branch-based development model. By isolating changes in separate branches, developers can innovate without affecting the stability of the main codebase, ultimately merging their work back into the main branch. This approach boosts productivity and provides a controlled environment for integrating changes.

However, despite these advantages, VCSs come with significant challenges [100]. One of the most critical issues in branch-based development is the occurrence of merge conflicts. These conflicts arise when changes in separate branches are incompatible, creating problems during the merge process. Developers often spend hours or even days detecting and resolving conflicts before successfully merging branches [66]. This issue is particularly pronounced in large teams, where multiple developers work on interconnected parts of the codebase.

Merge conflicts can be categorized into three types [101, 107]: textual conflicts, build conflicts, and test conflicts. Textual conflicts arise when two branches modify the same line(s) of code differently and are typically detected during the merge process. Git's text-based merging algorithm identifies these conflicts, which must be resolved before the merge can proceed. In contrast, build and test conflicts—often referred to as higher-order conflicts—are more complex and frequently go undetected by Git, as they do not involve direct overlaps in

the code. Build conflicts, in particular, remain hidden because the conflicting changes do not overlap directly. Specifically, build conflicts occur when the merged code fails to compile, while test conflicts arise when the code compiles successfully but fails to pass test cases.

Understanding the root causes of build conflicts is critical. These conflicts typically stem from contradictory changes made in different branches, such as alterations to dependencies, classes, or functions that lead to compilation failures. Diagnosing these issues requires deep knowledge of the code's structure and components, making them particularly challenging to resolve. The first step in addressing build conflicts is effective detection. While compilers can identify errors, they provide little insight into the root causes of conflicts. Therefore, there is a pressing need for studies that explore the scenarios in which build conflicts occur. These studies would inform the development of specialized tools for early detection, helping to prevent these conflicts from disrupting the development pipeline.

Resolving build conflicts is another critical challenge. Effective resolution requires a thorough understanding of the conflict's origin and context, so that appropriate fixes can be applied. This process can be intricate and time-consuming, as it often involves significant changes to the code to ensure that the merged branches are compatible. The development of advanced tools and strategies for build conflict resolution is essential to reduce the manual effort involved and to minimize delays in the development process.

Given the complexities of resolving build conflicts, the rise of Large Language Models (LLMs) in software development offers a promising avenue. With their ability to understand and generate code, LLMs could automate the resolution of build conflicts, addressing challenges like the difficulty of diagnosing conflicts and the manual effort involved in fixing them. Exploring the potential of LLMs in this context is essential, as they could lead to innovative tools that enhance the efficiency and reliability of managing build conflicts.

1.1 Thesis Goals

Given the heavy reliance on Version Control Systems (VCSs) in software development, it is crucial to explore methodologies for effectively detecting and resolving build conflicts. This thesis focuses on three key aspects: (i) Understanding the causes of build conflicts and developing a tool for their detection, (ii) Analyzing conflict resolution strategies and creating a framework to suggest effective resolutions, and (iii) Investigating the potential of Large Language Models (LLMs) to resolve build conflicts, including a comparison between different open source LLMs.

1.2 Thesis Organization

The following is the outline of the organization of this thesis:

- Chapter 2 discusses prior research on software merging, strategies for detecting and resolving merge conflicts, and the role of Large Language Models (LLMs) in program analysis and addressing software merge conflicts.
- Chapter 3 details the development and evaluation of BUCOND (Build Conflict Detector), of a graph-based pattern-matching tool designed to detect build conflicts via static analysis.
- Chapter 4 provides an overview of the design and evaluation of BuCoR (Build Conflict Resolver), a hybrid program transformation-based framework for resolving build conflicts.
- Chapter 5 describes the research into investigating the capability of Large Language Models (LLMs) to resolve build conflicts.

- Chapter 6 summarizes the key contributions and findings, and concludes the dissertation.

Chapter 2

Related Works

This chapter elaborates prior research on software merging, strategies for detecting and resolving merge conflicts, and the role of Large Language Models (LLMs) in program analysis and addressing software merge conflicts.

2.1 Automated Software Merge

Git, currently the most widely used Version Control System (VCS), utilizes a text-based merging approach [1]. This method works by comparing each line in the left and right branches with the base branch, flagging a conflict if both branches modify the same line differently. However, this approach is not foolproof. False positives can occur because intersecting edits do not always indicate an actual conflict. Furthermore, since Git's merging process does not account for the semantic meaning of the code, it can miss true conflicts, leading to false negatives [52]. In response to these limitations, the concept of speculative merging has been introduced. This technique aims to detect conflicts earlier in the development process by continuously pulling and merging all possible combinations of branches in the background [51, 63, 66]. However, speculative merging is often hampered by its complexity, making it less practical for large-scale projects.

Since unstructured merge treats source code as plain text, it struggles to handle conflicts effectively. To address some of these limitations, structured and semi-structured merging

approaches have been introduced [42, 44]. These methods aim to improve merge accuracy by considering the structure of the code. [52] conducted a large-scale empirical study, analyzing 30,000 merges from 50 open-source projects, to compare current structure-aware merge techniques. However, these approaches still face challenges, particularly in handling software refactoring. To mitigate some of these issues, [87] proposed a graph-based, refactoring-aware approach that attempts to understand code changes during the merge process while accounting for potential refactoring. However, the evaluation of this method was limited to merge conflicts caused by refactoring, which may limit its generalizability, as [75] reports that only 22

2.2 Conflict Awareness

Several approaches have focused on developing tools that visually represent the local changes made by developers on a shared platform to improve communication and collaboration among team members [45, 67, 86]. While these tools are not designed to directly detect conflicts, they contribute to conflict prevention by highlighting areas of the code that are prone to errors. For instance, [74] adopts a similar strategy, where files modified by multiple pull requests simultaneously are flagged. This proactive measure reduces the chances of conflicting changes being merged, thereby helping to prevent potential bugs or conflicts after the merge.

A variety of tools [45, 50, 51, 66, 67, 74, 86] have been developed to monitor and analyze the development activities of programmers, enhancing team awareness and coordination. For example, Palantír [86] provides developers with notifications about changes made by their peers, assesses the impact of these changes, and presents this information visually. Similarly, Cassandra [66] aims to minimize conflicts by tracking dependencies such as super-

sub and caller-callee relationships between program entities. By treating these dependencies as constraints, Cassandra identifies potential conflicts when tasks are performed in parallel and suggests conflict-free development sequences. However, these tools do not focus on localizing merge conflicts.

2.3 Empirical Studies

Several studies have tried to study correlations between merge conflicts and developer edits to identify potential indicators and refactoring types that make certain scenarios more conflict prone [40, 59, 70, 75]. However, these studies focus only on textual conflicts and do little to analyze the specific cause or resolution pattern. Based on these shortcomings, other studies have focused on analyzing textual conflicts though the categorization of their specific causes and/or resolution strategies [47, 62, 68, 107]. In particular, [68] reports that git-merge can falsely report textual conflicts when concurrent edits are made to adjacent lines of code (as opposed to the same line). In such scenarios, any approach that focuses on compiling a program to reveal higher-order conflicts will fail until all textual conflicts are resolved. In addition to categorization of the specific causes and resolution patterns of merge conflicts, [93] also includes higher-order conflicts in their analysis.

2.4 Merge Conflict Detection and Resolution

Tools were built to detect or resolve merge conflicts [5, 42, 43, 50, 51, 52, 56, 63, 69, 77, 88, 96, 111]. For instance, FSTMerge [5, 43, 52] 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 [42] 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 [53]. AutoMerge [111] 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 [56] uses deep learning to resolve textual conflicts. However, these tools do not detect higher-order conflicts.

SafeMerge [96] 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 [97] 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 [101]. 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.

Machine learning (ML)-based approaches for conflict detection have been developed using nine lightweight feature sets extracted from Git [83]. However, these methods fall short in providing useful suggestions for developers to resolve conflicts, and their evaluation processes are not very promising, as they often produce a high rate of false positives. Techniques involving program synthesis, deep learning, and version space algebra have been explored [56, 84, 111]. For instance, [84] uses a combination of program synthesis and a novel domain-specific language (DSL) to learn repeated resolution patterns in large C++ projects. Simi-

larly, [55] introduces a prototype approach that analyzes build logs to identify the category of build conflict and provide fixed resolution patterns. On the other hand, [56] presents a data-driven approach leveraging deep learning to generate conflict resolutions, applicable to any target language. However, this method primarily focuses on textual conflicts and requires a large dataset for effective model training.

2.5 Large Language Models

LLMs have a wide range of applications in software engineering, as explored in a systematic review by [64]. The capabilities of LLMs in performing tasks such as program repair, code generation, and code summarization have been thoroughly studied by [95], [98], and [106]. Additionally, [80] investigates the interpretability of LLMs to better understand their strengths and limitations in code analysis. The potential of LLMs to transform software engineering practices is further discussed by [61] and [81].

Researchers have also begun to explore the use of LLMs for resolving software merge conflicts. [94] explores a two-phase approach to resolving textual conflicts in software merges. They propose a two-stage method where ML is first used to predict resolution strategies, and then the output is utilized by an LLM, specifically ChatGPT [82], to address more complex conflict cases. Similarly, [85] examines the use of ChatGPT for resolving test conflicts. Their study investigates how well ChatGPT can identify test conflicts and understand both the origin and impact of these conflicts. [108] proposes a GPT-3 [49] based approach capable of resolving both textual and higher-order conflicts. However, the number of conflict categories considered is quite limited. While the use of LLMs has gained traction in the broader field of program analysis, their application in the area of software merging is still relatively limited.

Chapter 3

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

3.1 Motivation

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 [66].

A typical **merging scenario** in software version history involves four program commits: the base $\boldsymbol{b}$, left version $\boldsymbol{l}$, right version $\boldsymbol{r}$, and developers' merge result $\boldsymbol{m}$ (see Figure 3.1). Between l and r, there can be three types of merge conflicts [51, 101]: 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 si-

multaneously 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.

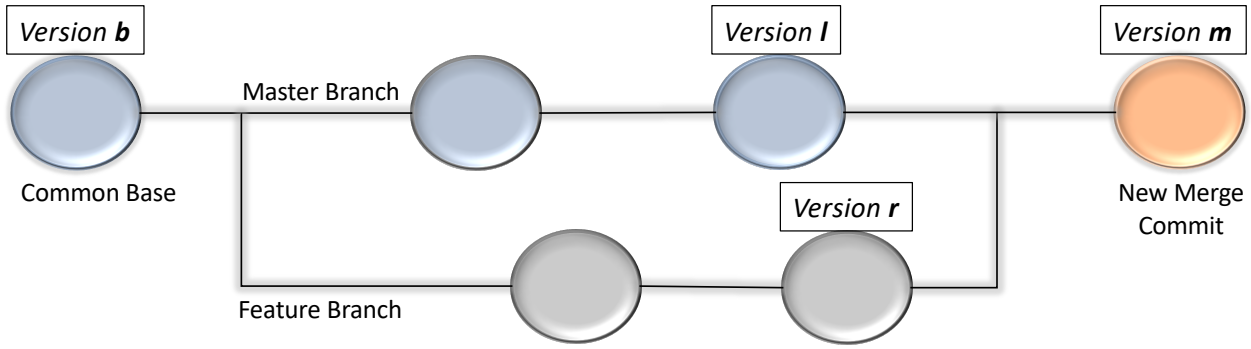

Figure 3.1: An exemplar merging scenario

Table 3.1: 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 [5, 42, 43, 50, 51, 52, 69, 77, 88, 96, 102, 111]. As shown in Table 3.1, majority of the tools target textual conflicts. Crystal [51] and WeCode [63] are among the few tools that detect all types of conflicts. They apply textual-merge of version control systems (e.g., git-merge) to tentatively merge two branches into one version $\boldsymbol{A_m}$, 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,

Changes in *l*	Changes in *r*

public class A {
public void foo() {
+ C.m();
... } } ...
public class C {
public static void m() {...}
 	
}
	public class A {
	public void foo() {
	...
	} } ...
	public class C {
	- public static void m() {...}
	+ public static void m(int p) {...}
	}

Figure 3.2: An exemplar build conflict.

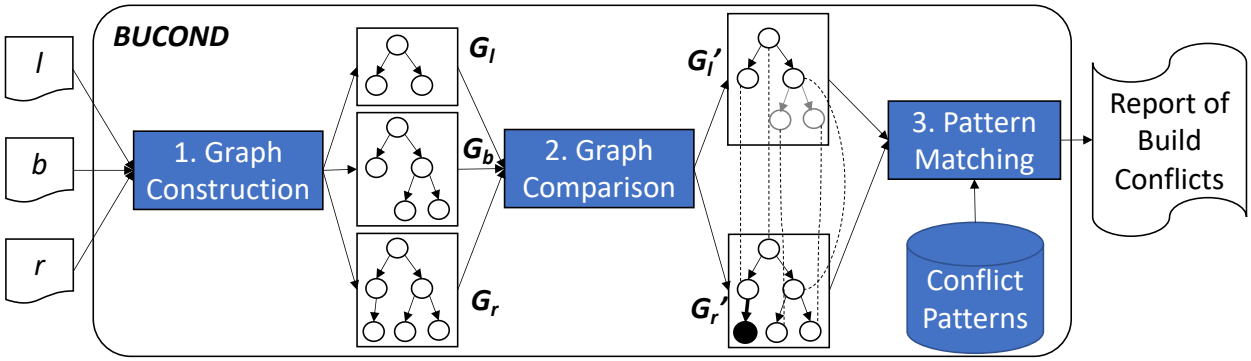

Figure 3.3: BUCOND comprises three phases.

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 [66], 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 3.3 (Section 3.3) for the complete list of program entities. Figure 3.2 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 3.3, 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.

3.2 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 [11], spring-cloud-alibaba [21], druid [9], 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.

As shown in Table 3.2, 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

Table 3.2: 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

def-use constraints include:

- When an entity is referenced, there should always be a corresponding entity definition visible to the reference.
- No entity should be defined multiple times, except for method overriding.
- When a sub-class implements an interface, the class should implement all methods declared by the interface.
- 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.

3.3 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 3.3, 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 3.3.1–3.3.3 explain all phases in detail.

3.3.1 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,

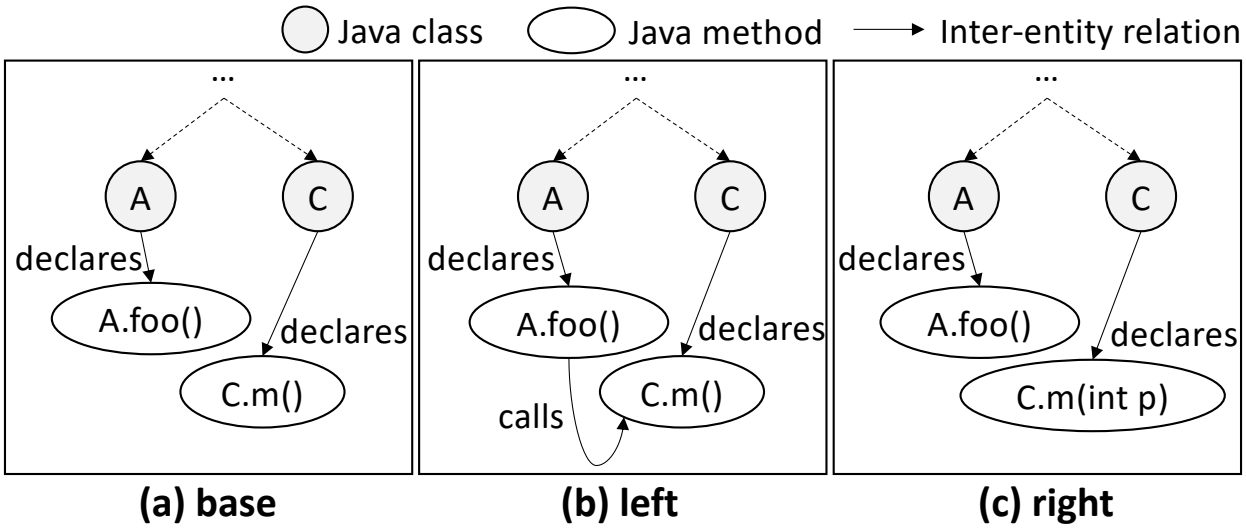

Figure 3.4: The PEGs for the merging scenario in Figure 3.2

renamed, or moved files) when modeling PEGs. This is because a commit often edits a small portion of files [46, 65, 109]. 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 [29] to create abstract syntax trees (ASTs). Based on the ASTs, Bucond extracts entities as well as inter-entity relations, and uses JGraphT [78] to build PEGs. In each PEG, *vertices* represent *entities* and *edges* show *inter-entity relations*. Figure 3.4 shows three exemplar PEGs for the merging scenario of Figure 3.2. In Figure 3.4, each node records both the type and fully qualified name of an 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 3.3). Specifically given entities E_1 and E_2,

- $\boldsymbol{E_1}$ **contains** $\boldsymbol{E_2}$ means the file folder of E_1 includes the file (or folder) of E_2.

Table 3.3: 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)	-	-

- ***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 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 3.3.3). 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.

3.3.2 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.

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 3.4, 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 $C.m(...)$ across graphs because the right branch r updated the method signature.

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:

$$Context\_Sim = \frac{N_1 \cap N_2}{N_1 \cup N_2} \tag{3.1}$$

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 [88] 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. Firstly, $Context\_Sim$ 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 im-

plementation can help further differentiate the same-context entities. Therefore, we defined three additional formulas to compute the similarity scores for the four entity types:

$$Function\_Sim = (Name\_Sim + Context\_Sim + Body\_Sim)/3 \tag{3.2}$$

$$Field\_Sim = (Name\_Sim + Type\_Sim + Expression\_Sim)/3 \tag{3.3}$$

$$EC\_Sim = (Name\_Sim + Context\_Sim + Expression\_Sim)/3 \tag{3.4}$$

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 [88] and shows great effectiveness in experiments.

$Body\_Sim$ describes how similar two method (or constructor) bodies are to each other. We reused GumTree [60] 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, **JCond** 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 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.

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 3.4, 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.

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 3.5), 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

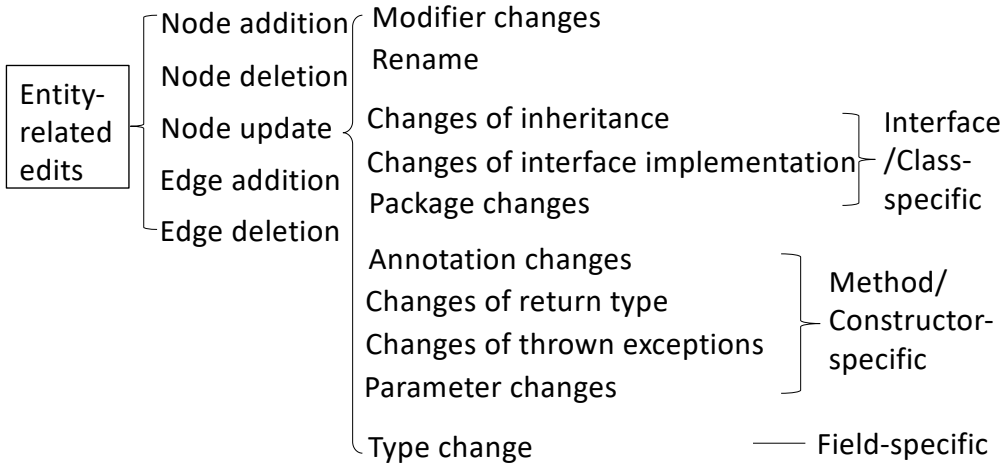

Figure 3.5: The types of edits that Bucond recognizes

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

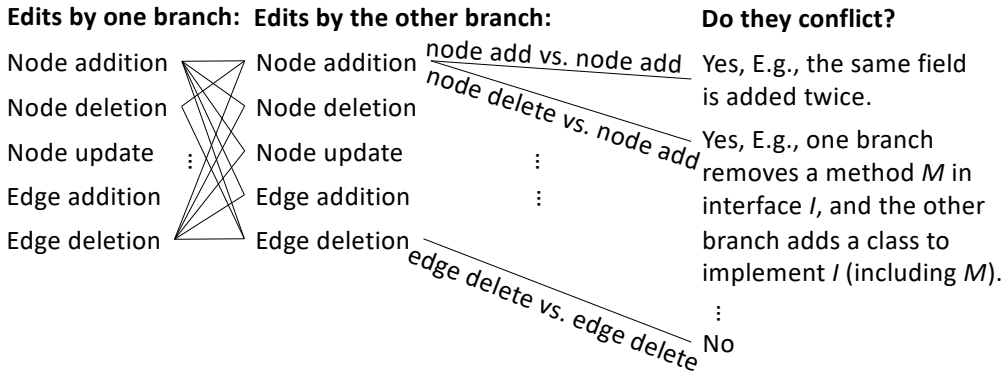

Figure 3.6: Our exploration procedure of conflict patterns

denote the linked revised graphs with G'_l and G'_r.

3.3.3 Phase III: Pattern Matching

Our research novelty mainly lies in this phase. We defined a pattern set to comprehensively enumerate the possible cases where the combination of branch edits can trigger build conflicts (Section 3.3.3). Accordingly, BUCOND performs pattern matching on the edits embedded in G'_l and G'_r to detect conflicts (Section 3.3.3).

Pattern Definition

Our preliminary study in Section 3.2 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 3.5, assessed whether a build error can occur for each combination, and defined conflict patterns for all recognized combinations that can trigger build errors.

Figure 3.6 visualizes our exploration process. We enumerated edit combinations between branches to decide whether any combination can trigger build errors. To recognize the

Table 3.4: The 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`.

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 3.2. 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 3.4 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 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 3.4 only shows a subset of all patterns in **BUCOND**. These exemplar patterns do not overlap with the ones shown in Table 3.2, although both sets are covered by **BUCOND**. Please refer to our open-sourced dataset for more details of the patterns.

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 3.5, 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.

Table 3.5: 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 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
```

3.4 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.

3.4.1 Datasets

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

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.

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 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 [6], Ant [38], 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 3.6 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.

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,

Table 3.6: 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

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.

To find conflicts manually, we randomly picked nine popular open-source Java repositories: Activiti [8], pebble [19], fastjson [11], vectorz [23], nuxeo [18], wildfly [25], webmagic [24], truth [22], and elasticsearch [10]. 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 [2], 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 3.7 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.

Table 3.7: 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

3.4.2 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 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.

3.4.3 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.*

3.4.4 Experiment Results on Dataset 2

Table 3.8: The merging scenario where BUCOND missed two build conflicts [33]

Changes in *l*	**Changes in *r***

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

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 3.8, 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 3.9: The merging scenario where Bucond missed a build conflict related to an `import`-declaration [7]

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 3.9, 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. This means that* Bucond *can identify build conflicts with high precision and high recall.*

As a software merge tool, IntelliMerge [88] 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 3.1, **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 better support for build-conflict detection.*

3.4.5 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 3.9. 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 3.10 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.

Table 3.10: The time cost of BUCOND when applied to the 13 merging scenarios in Dataset 3

Idx	**# of Analyzed Files**			**# of Analyzed Entities**			**Time Cost (minutes)**
	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

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.

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.*

3.5 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 3.4.1). 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 [88] 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.

3.6 Summary

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.

Chapter 4

Combining Example-Based and Rule-Based Program Transformations to Resolve Build Conflicts

4.1 Introduction

Version control systems (VCSs) like Git are popularly used in collaborative software development. With VCSs, programmers create and work on separate branches for feature addition, bug fixing, or feature improvement. Periodically, they merge branches into a primary branch, to integrate the edits applied to distinct branches into one program version. In such a scenario, **merge conflicts** can happen if the edits from different branches are incompatible. It is challenging and time-consuming to properly handle conflicts. Prior work shows that developers often spend hours or days detecting and resolving conflicts before correctly merging branches [66].

As illustrated in Figure 4.1, a typical merging scenario involves five program versions: two **branch versions** l and r whose edits need to be merged, **base version** b—the common origin of both branches, the **automatically merged version** A_m produced by git-merge when it naïvely integrates branch edits textually, and the **manually merged version** m

that developers create based on A_m. Among the various possible conflicts between l and r, **build conflicts** refer to the incompatible edits whose naïve integration triggers build errors in the resulting merged version. As shown in Figure 4.2, because l adds a call to $m()$ while r renames that method, the co-application of both edits can cause a compilation error of unresolved method reference. Namely, the newly added method call $m()$ is not associated with any defined method.

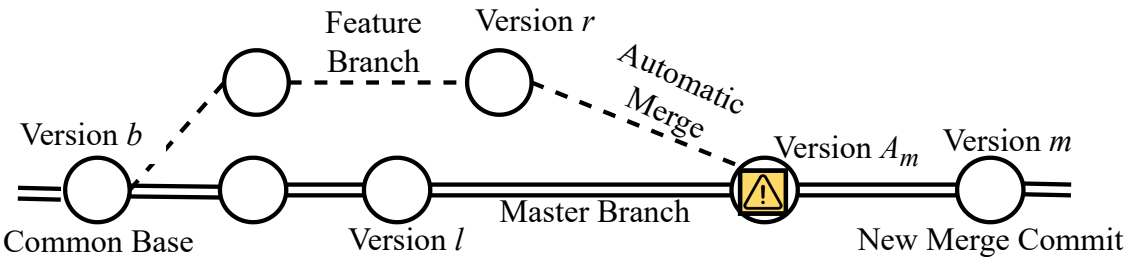

Figure 4.1: A typical merging scenario can involve up to five program versions

Although tools were created to detect build conflicts [99, 101], there is rare automatic support to resolve such conflicts. In particular, Gmerge [108] relies on Clang compiler messages to locate conflict-related changes in branches, and applies k-shot learning with GPT-3 to suggest symbol renaming for conflict resolution. MrgBldBrkFixer [97] starts with a naïvely merged version A_m, looks for build errors due to failed resolution of symbols, identifies symbol renaming changes in branches-to-merge, extracts related patches in either branch, and similarly applies those patches to fix build conflicts. These tools suffer from two limitations:

1. They only handle conflicts related to symbol renaming (Figure 4.2), but fail to resolve many other conflicts, such as those caused by class-hierarchy change, method addition/deletion, and parameter addition/deletion.

2. They suggest simple edits of symbol renaming to adapt *use*s of renamed entities, but fail to suggest complex edits that systematically change *uses* and surrounding context.

Changes in local (*l*) branch	Changes in remote (*r*) branch

public class A {
public void foo() {
+ C.m();
... } ... }
public class C {
public static void m() { ...
}
}
	public class A {
	public void foo() {
	...
	} ... }
	public class C {
	- public static void m() { ... }
	+ public static void init() {...}
	}

Figure 4.2: A build conflict due to the updated def of `m()` by r, and added use by l

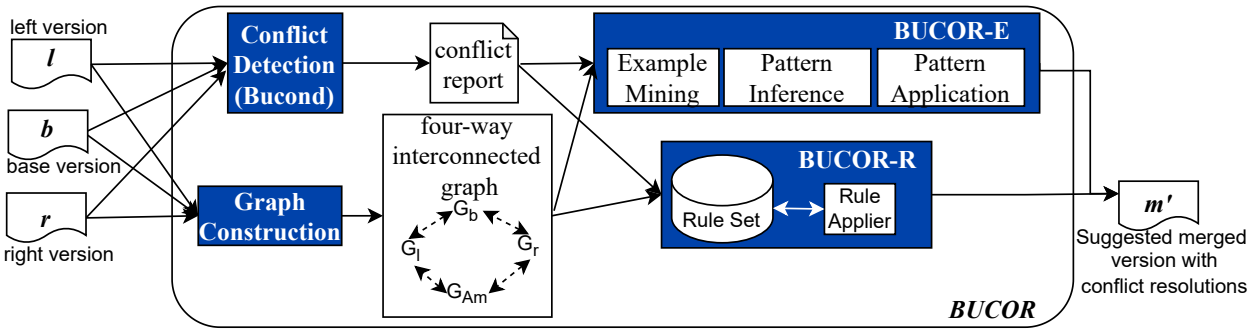

Figure 4.3: BuCoR leverages Bucond [99] to detect build conflicts, and employs two complimentary strategies to resolve conflicts

To overcome these limitations, we introduce BuCoR **(BUild COnflict Resolver)**, a novel resolver of build conflicts. As shown in Figure 4.3, given the three program versions (l, b, r) related to a merging scenario, BuCoR first applies Bucond [99], a static analysis-based tool, to reveal conflicting edits between branches. BuCoR also adopts git-merge to generate a naïvely merged version A_m, extending Bucond to (1) create a graphical representation for A_m and (2) relate that graph with those of the three input program versions. Based on the generated conflict report and four-way interconnected graph, BuCoR opportunistically applies two complementary strategies: BuCoR-E and BuCoR-R.

Example-based transformation (BuCoR-E): Prior work shows that many build conflicts are caused by mismatches between revised or removed *def* (short for "definition"), and newly

introduced *use* of the same program entities (e.g., classes and methods) [55, 91]. Inspired by this insight, BuCoR-E identifies the *def*-change responsible for each reported conflict, locates the owner branch, and mines that branch for any exemplar edit applied to adjust corresponding *use*s. It then infers a program transformation pattern from that edit. For each *use*-addition responsible for the reported conflict, BuCoR-E tentatively establishes context matching between the pattern and code in A_m. If a full or partial match is found, BuCoR-E customizes and applies the entire or partial pattern for conflict resolution.

Rule-based transformation (BuCoR-R): Prior studies reveal patterns frequently applied to resolve certain kinds of build conflicts [97, 101]. Following those studies, we defined and implemented 16 resolution rules/patterns in BuCoR-R. Given a conflict, BuCoR-R searches its rule set for an applicable rule; when a rule is found, BuCoR-R customizes that rule for edit application.

To evaluate the tool effectiveness, we applied BuCoR to 88 real-world build conflicts in 30 open-source projects. BuCoR correctly resolved 34 conflicts. BuCoR-E and BuCoR-R separately generated resolutions for 28 and 51 conflicts; 21 and 20 of those resolutions separately match developers' resolutions recorded in version history. All these numbers demonstrate BuCoR's effectiveness in handling conflicts. To sum up, we made the following contributions:

- We created BuCoR, a new static analysis-based resolver of build conflicts, to novelly combine example-based transformation with rule-based transformation.
- We explored BuCoR-E, an advanced example-based approach of conflict resolution. Different from existing tools, it applies program dependency analysis to derive resolution patterns from exemplar edits, establishes context matching for partial/full pattern customization as well as application, and ranks candidate resolutions to suggest the

Edits from the branches-to-merge

(a) Conflicting edits between branches

Changes in l (responsible def updates):
In `TypeSerializerConfig.java`, names of the file, Java class, and constructor are all updated to `SerializerConfig`.

Changes in r (responsible use introduction):
In a newly added file `XmlClientConfigBuilder.java`, a method is defined to access `TypeSerializerConfig`

```
+ private void handleSerializers(Node node,
      SerializationConfig serializationConfig) {
+   ...
+   if ("type-serializer".equals(name)) {
+    TypeSerializerConfig typeSerializerConfig = new
      TypeSerializerConfig();
+    typeSerializerConfig.setClassName(value);
+    final String typeClassName = getAttribute(child, "
      type-class");
+    typeSerializerConfig.setTypeClassName(typeClassName);
+    serializationConfig.addTypeSerializer(
      typeSerializerConfig);
+  } ... }
```

(b) BUCOR mines branch edits for exemplar edit E to adapt code

Edit example in l to adapt usage of `TypeSerializerConfig`:

```
private void handleSerializers(Node node, ...) {
  ...
- if("type-serializer".equals(name)){
-   TypeSerializerConfig typeSerializerConfig = new
      TypeSerializerConfig();
-   typeSerializerConfig.setClassName(value);
+ if("serializer".equals(name)){
+   SerializerConfig serializerConfig = new
      SerializerConfig();
+   serializerConfig.setClassName(value);
    final String typeClassName=retrieveAttribute(child, "
       type-class");
-   typeSerializerConfig.setTypeClassName(typeClassName);
-   serializationConfig.addTypeSerializer(
      typeSerializerConfig);
+   serializerConfig.setTypeClassName(typeClassName);
+   serializationConfig.addSerializerConfig(
      serializerConfig);
```

Tool-generated edits that are applicable to A_m, to resolve conflicts

(c) A naïve resolution producible by existing tools and BUCOR-R

```
private void handleSerializers(Node node, ...) {
  ...
  if ("type-serializer".equals(name)) {
- TypeSerializerConfig typeSerializerConfig = new TypeSerializerConfig();
+ SerializerConfig typeSerializerConfig = new SerializerConfig();
   typeSerializerConfig.setClassName(value);
   final String typeClassName=getAttribute(child, "type-
       class");
   typeSerializerConfig.setTypeClassName(typeClassName);
   serializationConfig.addTypeSerializer(
       typeSerializerConfig);
 } ... }
```

(d) A more comprehensive resolution produced by BUCOR-E

```
private void handleSerializers(Node node, ...) {
  ...
- if ("type-serializer".equals(name2)) {
- TypeSerializerConfig typeSerializerConfig = new TypeSerializerConfig();
- typeSerializerConfig.setClassName(value);
+ if ("serializer".equals(name2)) {
+ SerializerConfig serializerConfig = new SerializerConfig();
+ serializerConfig.setClassName(value);
   final String typeClassName=getAttribute(child, "type-
       class");
- typeSerializerConfig.setTypeClassName(typeClassName);
- serializationConfig.addTypeSerializer(typeSerializerConfig);
+ serializerConfig.setTypeClassName(typeClassName);
+ serializationConfig.addSerializerConfig(serializerConfig);
 } ... }
```

Figure 4.4: A motivating example of a build conflict whose resolution requires context-specific edits beyond symbol renaming

best one.

- We explored **BUCOR-R**, a rule-based approach to resolve frequently occurred conflicts in 16 conventional ways. No prior work implements such an approach.

- We systematically evaluated **BUCOR** with a dataset of 88 conflicts that span 21 types, and observed novel phenomena.

4.2 A Motivating Example

To facilitate discussion, this section introduces a running example we crafted based on a real-world project hazelcast [13]. As shown in Figure 4.4(a), a merging scenario has l rename a class `TypeSerializerConfig` and r insert code to use the original class. The naïve integration of these edits can trigger a build error as the newly introduced *use*s refer to a nonexistent class *def*; thus, a build conflict occurs.

To resolve such a conflict, existing tools simply update class references by replacing `TypeSerializerConfig` with `SerializerConfig` (Figure 4.4(c)). However, such a replacement is insufficient, as the context still has variable `typeSerializerConfig` and literal "`type-serializer`" match the original class usage. Consequently, after existing tools update class references, developers need to manually replace *def*s and *use*s of related variables/literals, to ensure consistent updates and prevent semantic conflicts. Furthermore, when such class *use*s are introduced at multiple places, developers have to go over all places to manually apply those edits again and again, which process is tedious and error-prone.

To overcome the limitations of existing tools and better help developers, we introduce BuCoR—a hybrid approach to combine example-based resolution with rule-based resolution. The example-based resolution BuCoR-E is derived from our insight, on the association between build errors in branches and build conflicts in the merged version. Basically, many build errors are caused by mismatches between *def* and *use* of the same program entities; many build conflicts are caused by mismatches between modified *def* and newly introduced *use* of entities [55, 91]. *If on either branch, developers resolved def-use mismatches by applying specialized edits; then they are likely to reapply the same or similar edits to resolve conflicts that show the same kind of mismatches in software merge.*

For each reported conflict, BuCoR-E locates the responsible *def*-change, and mines the con-

tributing branch for developers' exemplar edit E applied to adjust existing *use*s of the changed entity. As shown in Figure 4.4(b), the edit example is extracted from l by BuCoR, because that edit adjusts usage of `TypeSerializerConfig`—the renamed class. The edit E not only updates references to the old class/constructor, but also revises a related variable `typeSerializerConfig`, a literal "`type-serializer`", and a method call `addTypeSerializer(...)`. BuCoR-E then derives a transformation pattern P from E, by abstracting away irrelevant edit detail and/or program context.

For the responsible *use*-introduction of each conflict, BuCoR-E tentatively establishes context matching between P and the edit location. If the matching succeeds, BuCoR-E customizes P by generating edit operations with respect to the matched context. BuCoR then applies those operations to transform code. As shown in Figure 4.4(d), the context-to-handle is different from the original edit context (see Figure 4.4(b)): it uses a different variable (`name2` vs. `name`), and invokes a different method (`getAttribute(...)` vs. `retrieveAttribute(...)`). Albeit the differences, BuCoR still resolves the conflict by mimicking developers' coding practices.

In addition to BuCoR-E, our tool also integrates a rule-based transformation approach BuCoR-R. This is because there are scenarios where l and r do not have exemplar edit E in response to *def*-changes, making BuCoR-E less useful. BuCoR-R can opportunistically suggest resolutions when BuCoR-E does not work, to mitigate the limitation of example-based transformation. For the merging scenario described above, BuCoR suggests two alternative resolutions for developers to review (see Figure 4.4(c)–(d)).

4.3 Approach

As shown in Figure 4.3, BuCoR has four components: conflict detection, graph construction, BuCoR-E, and BuCoR-R. This section explains each of them in detail.

4.3.1 Conflict Detection (Bucond [99])

Bucond statically analyzes three program versions related to each merging scenario—b, l, r—to identify build conflicts. It models each version as a **program entity graph (PEG)**, which captures defined program entities (e.g., classes and methods) and their relations (e.g., type reference or method calls). By comparing PEGs, Bucond extracts entity-level edits in l and r. It then matches these edits with 57 predefined patterns of conflicting edits. For instance, one pattern checks when a method is renamed in one branch, whether the other branch adds any call to the original method. Bucond reports a conflict if any match is found. Unlike compiler-based tools, Bucond does not try to generate or build the naïvely merged version A_m. It can detect conflicts even if A_m is uncompilable, and pinpoint the specific edits responsible—which compilers cannot do.

For implementation, Bucond uses JavaParser [29] to parse source code, and adopts JGraphT [30] to construct and analyze PEGs.

4.3.2 Graph Construction

To facilitate users' conflict comprehension and our tool's resolution placement, we extended Bucond to construct a **four-way interconnected graph**. Our extension involves two parts: (1) generating a merged version A_m and (2) comparing its PEG with those of given program versions. As a first step, BuCoR uses the widely used tool git-merge [39] to create a naïvely

merged version A_m. While git-merge can detect textual conflicts—cases where multiple branches apply divergent edits to the same code fragment, it does not detect or resolve build conflicts. The A_m it produced offers a program context for which **BuCoR** later suggests resolution edits.

In Step 2, to relate A_m with l and r, **BuCoR** also creates a PEG for A_m, and matches this graph against the PEGs of other versions. As shown in Figure 4.5, we denote the PEGs as G_b, G_l, G_r, G_{Am}, corresponding to the base, left, right, and naïvely merged versions. The differences between G_l and G_b, and between G_r and G_b, are represented as $\Delta(G_l, G_b)$ and $\Delta(G_r, G_b)$; they capture entity-level edits like entity addition/deletion/update, and relation addition/deletion. They are computed by Bucond.

We use $\cap(G_{Am}, G_l)$ and $\cap(G_{Am}, G_r)$, to separately denote intersections between G_{Am} and G_l, and between G_{Am} and G_r. They represent structural commonality between versions, serving as anchors to align program context as well as edits between A_m, l, and r. **BuCoR** recognizes such commonality using the graph-matching algorithm from Bucond. Intuitively, given two graphs under comparison, the algorithm first identifies exact node matches based on entity types and fully qualified names (FQNs). For nodes that remain unmatched, it ambiguously matches nodes based on the similarity of their internal code implementation and surrounding context. The identified common elements across program versions enable **BuCoR** to accurately position edits during conflict resolution.

4.3.3 Example-based Transformation (BuCoR-E)

To resolve a conflict, **BuCoR-E** operates in three phases. It first mines branches for edit example(s), infers a transformation pattern from each example, and then customizes as well as applies those patterns to revise A_m.

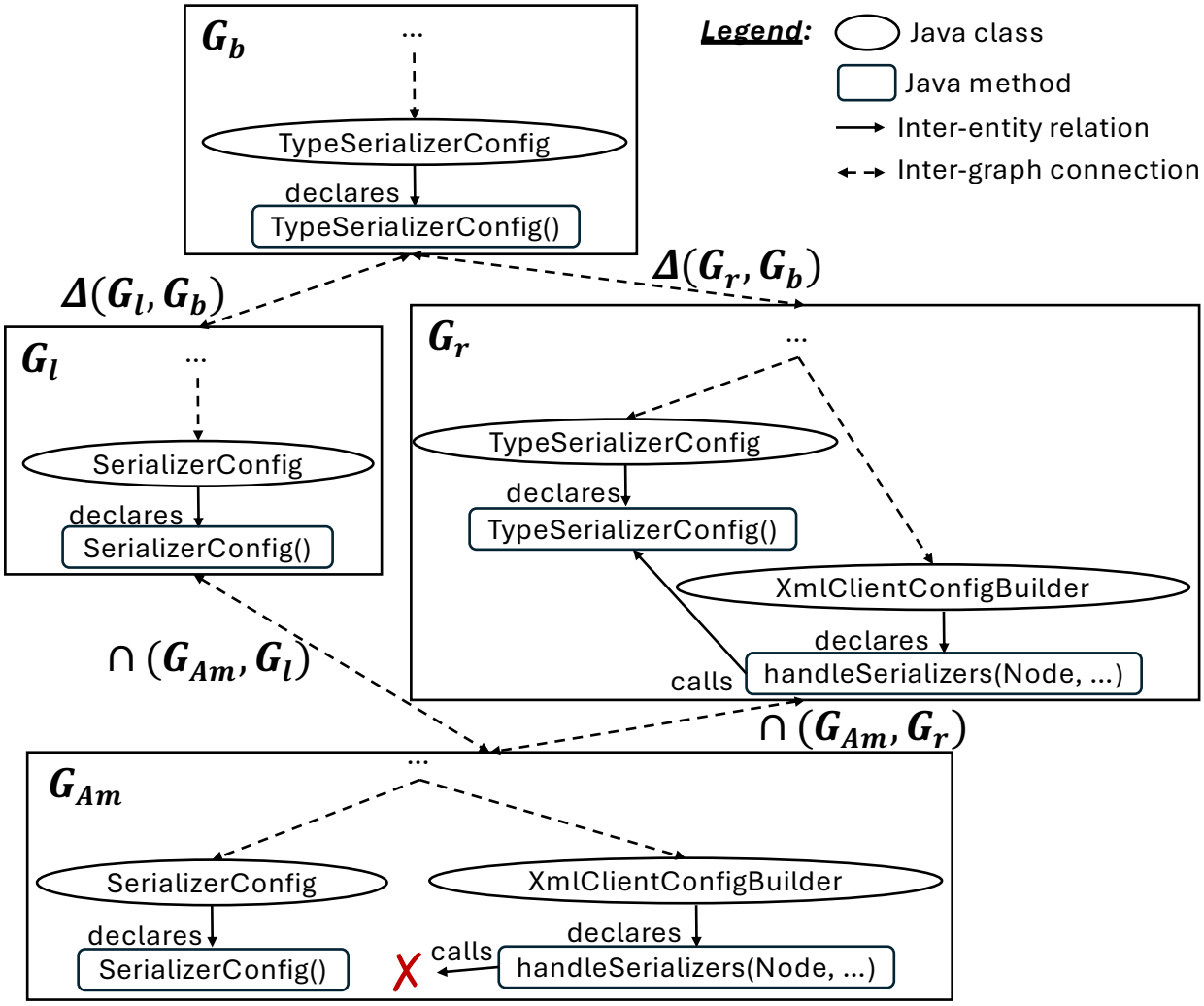

Figure 4.5: Four-way interconnected graph for the motivating example in Figure 4.4

Phase I. Example Mining

Given a conflict, BuCoR-E identifies the responsible *def*-change, mines the branch contributing that change, and looks for any exemplar edit which adapts entity *use*s and surrounding code for that change. In our motivating example, because l renames class `TypeSerializerConfig` and its constructor, it is the contributor branch of *def*-change. By traversing entity-level edits of l—$\Delta(G_l, G_b)$, BuCoR-E locates changes of the corresponding class/constructor references. Namely, if an entity $\boldsymbol{e}$ uses `TypeSerializerConfig` in b but uses `SerializerConfig` in l, then the entity e contains an edit example. Certainly, if multiple entities adapt their usage to the same *def*-change, BuCoR-E considers all these entities to have relevant examples.

To extract and represent each edit example, BuCoR-E applies an off-the-shelf syntactic differencing tool GumTree [60] to the base and branch versions of e, to generate an edit script of Abstract Syntax Tree (AST). The script may have four kinds of operations:

- `update(`t`, `v_n`)`: To replace the old value of node t with the new value v_n.
- `add(`t`, `t_p`, `i`, `l`, `v`)`: To add a new node t to the AST, as the i^{th} child of node t_p. Here, l and v separately specify t's entity type (e.g., method invocation) and its value (e.g., the statement string).
- `delete(`t`)`: To remove a node t from the AST.
- `move(`t`, `t_p`, `i`)`: To move a node t and its subtree to the i^{th} child of node t_p.

For our motivating example, the script extracted from the located exemplar edit includes eight operations: each operation updates a node to replace a string literal, type usage, constructor/method usage, or variable usage (see Figure 4.6).

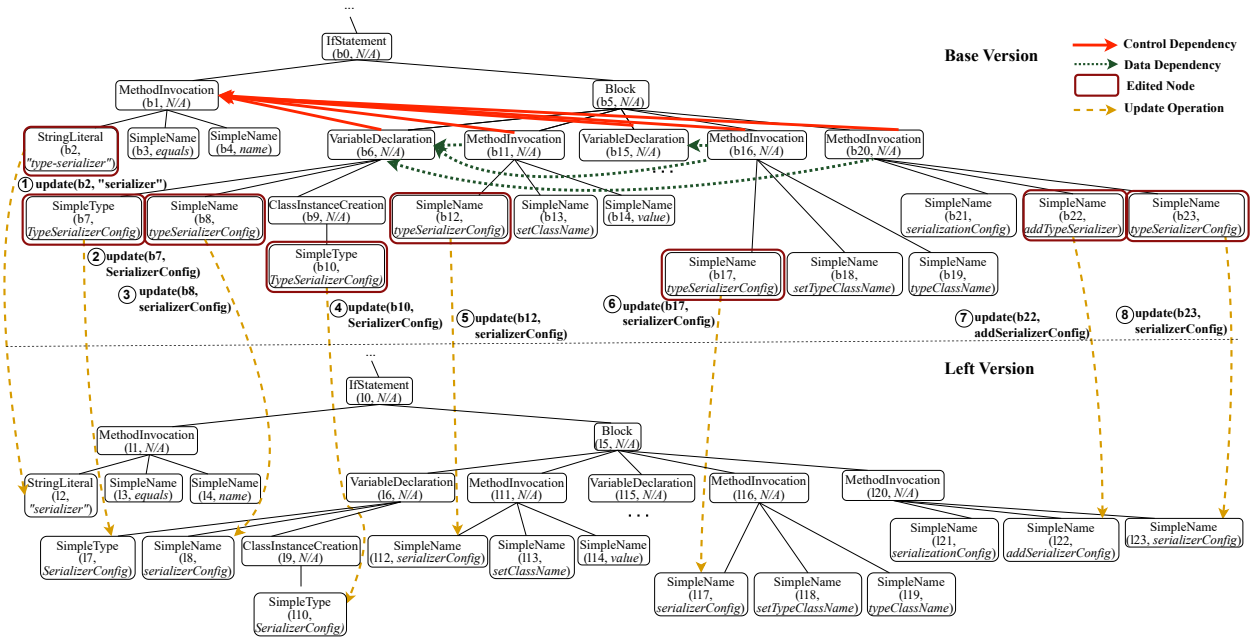

Figure 4.6: From the edit example shown in Figure 4.4(b), eight AST edit operations are extracted (see ①–⑧)

Phase II. Pattern Inference

In each example E, not every edit operation was applied to adapt *use*s for a given *def*-change, and not all unchanged code is relevant to those adaptive changes. To infer a minimal transformation pattern P, BuCoR-E takes two steps: edit refinement and context refinement.

Step 1 of Phase II: Edit Refinement. BuCoR-E extracts edit operations related to *use*s for a given *def*-change, but abstracts away irrelevant operations. Specifically, if a *def*-change revises an existing field or method, BuCoR-E considers all accesses of that field/method as *use*s. If a *def*-change revises a class, BuCoR-E considers (1) all accesses to that class and (2) all accesses to the variables instantiated with that class as *use*s. For our motivating example, as shown in Figure 4.6, edit operation ② is relevant as it updates usage of the changed class `TypeSerializerConfig`; ④ is relevant as it updates usage of the changed constructor

`TypeSerializerConfig`; operations ③, ⑤, ⑥, and ⑧ are relevant as they all update usage of variable `typeSerializerConfig`—an instance of `TypeSerializerConfig`.

Starting with the initial set of *use*-related operations (E_0), **BuCoR-E** further includes operations that occur within the same statement(s) as any operation in E_0. For instance, operation ⑦ gets incorporated because it appears in the same statement as ⑧. Our rationale is that when two edit operations are applied to the same statement, they are very likely to be semantically related. To conservatively preserve the completeness of extracted edits, we include all operations co-applied with *use*-related operations in the same statements. We denote the expanded operation set with E_1.

As shown in Figure 4.6, **BuCoR-E** locates edited nodes for the entire edit script in base b, identifies the subset of edited nodes covered by E_1, and conducts static analysis to find other edited nodes directly depended on by the subset. **BuCoR-E** extracts two kinds of statement-level dependencies:

- **Control Dependency:** x depends on y, if whether or not x is executed depends on the execution outcome of y.
- **Data Dependency:** x depends on y if x uses a variable defined by y.

Such dependencies are essential to reveal changes that constrain any *use*-change. Namely, without the operations on which E_1 depends, E_1 can wrongly modify semantics and introduce errors like accessing undefined variables. **BuCoR-E** adopts WALA [26], a widely used program analysis tool, to conduct dependency analysis. By extracting *use*-related operations and any operations they depend on, **BuCoR-E** ensures to abstract away non-essential co-applied edit. For our motivating example, ① also gets included, as the `if`-condition controls whether or not edited nodes in E_1 get executed.

Step 2 of Phase II: Context Refinement. BuCoR-E extracts the minimal subtree that covers all refined operations, to abstract away non-essential context and derive a conflict-resolution pattern. Listing 4.1 shows the pattern BuCoR-E infers from Figure 4.4(b). Compared with the original example, this pattern is more concise. It has all operations necessary to adapt the usage of an updated class and its constructor. Meanwhile, it removes surrounding program context of the `if`-statement, and an unchanged statement in that structure: `final String typeClassName=getAttribute(child, "type-class");`.

Listing 4.1: A pattern inferred from the mined edit example

```
1 - if(captionpos"captionpostypecaptionpos-captionposserializercaptionpos".equals(name
    )){
2 -   TypeSerializerConfig typeSerializerConfig = new TypeSerializerConfig();
3 -   typeSerializerConfig.setClassName(value);
4 -   typeSerializerConfig.setTypeClassName(typeClassName);
5 -   serializationConfig.addTypeSerializer(typeSerializerConfig);
6 + if(captionpos"captionposserializercaptionpos".equals(name)){
7 +   SerializerConfig serializerConfig = new SerializerConfig();
8 +   serializerConfig.setClassName(value);
9 +   serializerConfig.setTypeClassName(typeClassName);
10 +  serializationConfig.addSerializerConfig(serializerConfig);
```

Phase III: Pattern Application

To resolve conflicts, BuCoR-E enumerates patterns inferred for each conflict-responsible entity *use*, tentatively matching each pattern P with the program context to see whether the pattern can be customized and applied. This phase has two steps: context matching and AST rewriting.

Algorithm 3: The context-matching algorithm

Input: Pattern P, method-to-edit M, changed entity e
Output: Node matches between the ASTs: W

```
/* Find a match for the last usage of e                                    */
```

3.1 Locate a statement $s_p \in P$, which has the last usage of e
3.2 Search among M for a statement s_m that best matches s_p
3.3 **if** *match is found* **then**
3.4 $W.add(s_p, s_m, matchingScore)$

```
    /* Use the matched nodes as anchors to guide tentative matches for siblings
       */
```

3.5 **foreach** *sibling statement* l_p *coming before* s_p **do**
3.6 **foreach** *sibling statement* l_m *coming before* s_m **do**
3.7 Search for a best match in a backward manner
3.8 **if** *a best match is found* **then**
3.9 $W.add(l_p, l_m, bestScore)$
3.10 **foreach** *sibling statement* l_p *coming after* s_p **do**
3.11 **foreach** *sibling statement* l_m *coming after* s_m **do**
3.12 Search for a best match in a forward manner
3.13 **if** *a best match is found* **then**
3.14 $W.add(l_p, l_m, bestScore)$

```
    /* Compare parent nodes of s_p and s_m                                  */
```

3.15 Locate the parent node of s_p, $p_p \in P$, if there is any
3.16 Locate the parent node of s_m, $p_m \in M$, if there is any
3.17 Check whether p_m matches p_p
3.18 Repeat 1.3–1.17 by treating p_p as s_p and treating p_m as s_m, until no more parent can be found or the parent match fails

Step 1 of Phase III: Context Matching. Because Phase II applies backward dependency analysis to *use*-related edited nodes to infer P, the resulting contextual AST typically includes critical nodes (i.e., edited nodes using entity e) as leaf nodes, and noncritical ones (i.e., unchanged nodes or edited nodes not using e) as inner nodes. Our goal is to adapt usage of e. Thus, to decide whether an inferred pattern is applicable, we prioritize the match search for critical nodes. Namely, if critical nodes do not match, a pattern is inapplicable even if noncritical nodes match. If critical nodes match, a pattern is at least partially applicable even if higher-level noncritical ones do not match. We designed an algorithm to perform *use*-centric, bottom-up context matching.

As illustrated by Algorithm 3, given a pattern P, a method M whose entity *use* causes a conflict, and the changed entity e, BuCoR-E tentatively matches nodes in a bottom-up manner with the best effort. Specifically, it first locates a statement s_p in P that contains the last usage of e, to search among M for a statement s_m best matching s_p (see 1.1–1.2). To compare statements, we check their AST node types and values to calculate a matching score as below:

- **Type Match:** Two nodes, x and y, have a type match if their node types are identical. Such a match increments the initial matching score (i.e., 0) by 1. Type match enables patterns to get applied to the context (1) distinct from the original edit example, but (2) preserving the structural consistency.

- **Value Match:** Two nodes, x and y, have a value match if their strings have a 3-gram similarity score greater than 0.618. Here, the string of a simple statement (e.g, `MethodInvocation`) includes all content of that statement; the string of a complex statement (e.g., conditional or loop structure) only includes the structure header (e.g., `if`-condition or `for`-loop header). A successful value match increments the overall

matching score by the calculated similarity value. This threshold-based approach allows similar nodes to match, even though they are not identical. Such a flexibility also increases the applicability of inferred patterns.

Particularly, if x and y are identical, they have a type match and the 3-gram similarity is 1, so the overall matching score is 2.

Once a best match of s_p is found (with score > 1.618), **BuCoR-E** treats the pair (s_p, s_m) as anchor points to guide node matching for their siblings (see 1.5–1.14). Specifically for siblings preceding s_p, it enumerates nodes in reverse order: it first compares s_p's nearest sibling against all siblings preceding s_m for a best match; it then proceeds to match s_p's more distant siblings against any remaining unmatched nodes, moving backward from the most recently matched one. Similarly, for siblings following s_p, **BuCoR-E** matches nodes in forward order.

After searching best matches for all sibling nodes of s_p, the process continues to tentatively match the parent node of s_p with that of s_m. If this match succeeds, **BuCoR-E** repeats 1.3–1.17 in Algorithm 3, using the newly matched blocks as anchors for further match. The procedure continues until no more parent node in P can be found or the parent match fails. In the end, the algorithm outputs best matches of nodes, and their matching scores.

If Phase II infers multiple patterns from edit examples, these patterns may be all applicable to the method-to-edit M. To find the best one, we defined two ways to score the overall context matching between each pattern and M, based on the output of Algorithm 3:

- **Sum of Matching Scores (Σ_m):** Adding up the scores of all best matches.
- **Count of Exact Matches (C_m):** Counting the number of exact matches (i.e., matching score = 2).

We believe that the greater the contextual similarity between a candidate pattern P_i and M, the easier it is to apply that pattern and the more likely the resulting version is correct. Thus, among all applicable patterns, we select the one with the highest Σ_m. If multiple candidates have a tie, we choose the one with the highest C_m, ensuring a closer semantic match between that pattern and M.

In our motivating example, the method-to-edit (Figure 4.4(c)) shares a highly similar context with the inferred pattern (Listing 4.1): $\Sigma_m = 9.94$ and $C_m = 4$. All five edited statements in the pattern sequentially find best matches within the method: four statements match exactly, and the `if`-condition has a highly similar counterpart.

Step 2 of Phase III: AST Rewriting. To apply the selected pattern to M, BuCoR-E manipulates the AST of M based on context-matching results, and pretty-prints the updated AST to suggest a resolved version to developers. Although BuCoR-E matches context using statement-level similarity calculation, it can manipulate ASTs at a finer granularity (e.g., updating a symbol). This is because the edit operations inferred from examples by GumTree are finer-grained; reapplying these operations in new context minimizes the modification and helps prevent unwanted changes. Furthermore, if the selected pattern only has partial context to match M, only the edit operations corresponding to the matched part get applied; the remaining ones corresponding to unmatched part are not applied. In this way, BuCoR-E resolves conflicts with the best effort, to maximize its applicability and conflict-resolution capability.

4.3.4 Rule-based Transformation (BuCoR-R)

BuCoR-R resolves conflicts via rule-based program transformations. To define practical resolution rules or patterns, we manually inspected the rules and data observed by prior empirical

Table 4.1: The 16 resolution rules used in BuCoR-R

Idx	Conflict Type	Resolution Pattern
C1	Class: rename def vs. add use	Update the added use
C2	Class: add method def in super vs. add sub class	Update method def in sub to match super
C3	Class: change a method's parameter list in super vs. add sub class	Update method def in sub to match super
C4	Class: change a method's return type in super vs. add sub class	Update method def in sub to match super
C5	Import: remove def vs. add use	Re-add the def (i.e., add back the removed entity import)
C6	Package: rename def vs. add use	Update the added use (i.e., update import with the new package name)
C7	Interface: rename def vs. add use	Update the added use
C8	Interface: add method def in super vs. add class to implement the super	Add/update method def in class to override the new method in super
C9	Interface: change a method's parameter list in super vs. add class to implement the super	Update parameter list in class to match super
C10	Interface: remove method def in super vs. add class to implement the super	Remove method def in class to match super
C11	Interface: rename a method def in super vs. add class to implement the super	Rename method def in class to match super
C12	Interface: change a class to implement the interface vs. change a method's return type in the class	Update return type of method in class to match super
C13	Field: rename def vs. add use	Update the added use
C14	Field: add def vs. add def	Remove redundant field definition
C15	Method: rename def vs. add use	Update the added use
C16	Method: add def vs. add def	Remove redundant method definition

studies [55, 97], and expanded those rule sets to generalize rules across similar but different conflict types. Table 4.1 shows the resulting 16 conventional patterns we implemented in BuCoR-R; each pattern resolves one type of frequent conflicts. Among the 16 conflict types, only 6 types involve symbol renaming (C1, C6, C7, C11, C13, C15). As prior work only resolves conflicts caused by symbol renaming, they cannot handle 10 of the conflict types BuCoR-R focuses on. Only 7 of the 16 patterns were mentioned by prior work; we similarly summarized 9 new patterns.

Table 4.2: 21 conflict types of the 88 conflicts in our dataset

Idx	**Conflict Type**	**# of Conflicts**
C1	Class: rename def vs. add use	7
C2	Class: add method def in super vs. add sub class	2
C3	Class: change a method's parameter list in super vs. add sub class	2
C4	Class: change a method's return type in super vs. add sub class	1
C5	Import: remove def vs. add use	5
C6	Package: rename def vs. add use	2
C8	Interface: add method def in super vs. add class to implement the super	6
C9	Interface: change a method's parameter list in super vs. add class to implement the super	1
C10	Interface: remove method def in super vs. add class to implement the super	1
C11	Interface: rename a method in super vs. add class to implement the super	2
C12	Interface: change a class to implement the super vs. change a method's return type in the class	9
C14	Field: add def vs. add def	3
C15	Method: rename def vs. add use	8
C16	Method: add def vs. add def	2
C17	Class: remove def vs. add use	10
C18	Constructor: change the parameter list vs. add use	5
C19	Field: change a field's type vs. add use	1
C20	Field: remove def vs. add use	8
C21	Method: change the parameter list vs. add use	4
C22	Method: change the return type vs. add use	1
C23	Method: remove def vs. add use	8

BUCOR-R has a Rule Applier to opportunistically resolve conflicts based on predefined rules. Namely, for each reported conflict, BUCOR-R tentatively matches the conflict against documented conflict types. If a match is found, Rule Applier applies helper functions to the responsible entity's *def* or *use*, to rewrite the AST nodes.

Table 4.3: The experiment result of BUCOR

	# of Resolutions Generated	**# of Correct Resolutions**	**Coverage (C)**	**Accuracy (A)**
BUCOR-E	28	21	32% (28/88)	75% (21/28)
BUCOR-R	51	20	58% (51/88)	39% (20/51)
BUCOR	79	41	74% (65/88)	52% (41/79)

4.4 Experiment

To evaluate BUCOR, we defined three research questions (RQs):

- **RQ1:** How often can BUCOR generate resolutions?
- **RQ2:** What percentage of generated solutions are correct?
- **RQ3:** What is the effectiveness comparison between BUCOR-E and BUCOR-R?

We conducted the experiment on a Windows machine with AMD Ryzen 9 8945HS @4.00GHz and 16 GB memory. We did not empirically compare BUCOR with Gmerge [108] or Mrg-BldBrkFixer [97], as they are close-sourced tools targeting C/C++ programs while BUCOR focuses on Java code. The following subsections describe our dataset, evaluation metrics, and results.

4.4.1 Dataset

We constructed our evaluation dataset by reusing the datasets of prior work [55, 99] with our best effort. Given a merging scenario with at least one known build conflict, we decided whether it is reusable based on the following criteria:

a) Both l and r build successfully.

b) The automatically merged version A_m output by git-merge does not contain any textual conflict.

c) The build conflict is detectable by Bucond—BuCoR's conflict detection module.

We ended up with a dataset of 88 build conflicts from 55 merging scenarios, which were mined from in total 30 open-source Java repositories. Most of the cases we filtered out do not satisfy a) or b). As shown in Table 4.2, the 88 conflicts belong to 21 types: 14 of the types overlap with those mentioned in **BuCoR-R** (see Table 4.1), the other types (C17–C23) are not resolvable by any of the predefined rules. For each conflict, we retrieved and inspected developers' merged version m, using it as the ground truth of conflict resolution.

4.4.2 Metrics

We defined two evaluation metrics:

Coverage (C) measures among all known conflicts, how many have at least one resolution generated by the resolver:

$$C = \frac{\text{\# of conflicts with at least one resolution generated}}{\text{Total \# of known conflicts}}$$

Accuracy (A) measures among all suggested resolutions, how many of them are correct:

$$A = \frac{\#\text{ of correct resolutions}}{\text{Total }\#\text{ of generated resolutions}}$$

A resolution is correct if the resolved version is semantically equivalent to the ground truth—developers' resolution recorded in m. Two authors independently inspected **BuCoR**'s resolution to examine semantic alignment, and had a discussion whenever they disagree.

4.4.3 RQ1: BuCoR Resolution Coverage

As shown in Table 4.3, **BuCoR-E** and **BuCoR-R** generated resolutions for 28 and 51 cases, respectively. The coverage metrics they separately achieved are 32% and 58%. When combining their outputs, **BuCoR** generated at least one resolution for 65 cases, resulting in an overall coverage of 74% (65/88). There are 14 cases where both strategies suggested resolutions. Among the remaining, **BuCoR-E** and **BuCoR-R** separately resolved 14 and 37 cases. These numbers imply the two resolution strategies to complement each other.

Listing 4.2: A conflict without edit example in branches

```
1 /* Left Version: PathItem.java adds a method call to setRef(...) */
2 + pathItemObject.setRef(apiCallback.callbackUrlExpression());
3
4 /* Right Version: Reader.java renames setRef(...) */
5 - public void setRef(String ref) {
6 + public void set$ref(String $ref) {
```

Listing 4.3: A conflict whose resolution is unconventional

```
1 /* Left Version: JedisCluster.java removes a field */
2 - private int timeout;
```

```
3
4 /* Right Version: JedisCluster.java adds a reference to field timeout */
5 + public Set<String> spop(final String key, final long count) {
6 +  return new JedisClusterCommand<Set<String»(connectionHandler, timeout,
      maxRedirections) {
7 +   @Override
8 +   public Set<String> execute(Jedis connection) {
9 +    return connection.spop(key, count);
10 +   } }.run(key); }
```

Listing 4.4: A conflict whose ground-truth resolution partially overlaps with exemplar edits [31]

```
1 /* Left Version moves a method between classes */
2 /* ClientMap.java */
3 - public final SerializationService getSerializationService() {...}
4 /* ClientContext.java */
5 + public SerializationService getSerializationService() {...}
6
7 /* Right Version: ClientMultiMapProxy.java adds calls to the original method
      */
8 + public boolean put(K key, V value) {
9 +  Data keyData = getSerializationService().toData(key);
10 +  Data valueData = getSerializationService().toData(value);
11 +  PutRequest request = new PutRequest(proxyId, keyData, valueData, -1,
      ThreadUtil.getThreadId());
12 +  Boolean result = invoke(request, keyData);
13 +  return result;
14 + }
```

Listing 4.5: A conflict whose ground-truth resolution has no overlap with exemplar edits [32]

```
1 /* Left Version: RedisClient.java adds a call to one of its constructors */
```

```
2 + public RedisClient(String host, int port, int connectTimeout, int commandTimeout) {
3 +  this(new HashedWheelTimer(), new NioEventLoopGroup(), NioSocketChannel.class,
     host, port, connectTimeout, commandTimeout);
4 + }
5
6 /* Right Version: RedisClient.java adds a parameter to one of its constructors
     */
7 - public RedisClient(final Timer timer, EventLoopGroup group, Class<? extends SocketChannel>
     socketChannelClass, String host, int port, int connectTimeout, int commandTimeout) {
8 + public RedisClient(final Timer timer, ExecutorService executor, EventLoopGroup group,
     Class<? extends SocketChannel> socketChannelClass, String host, int port, int
     connectTimeout, int commandTimeout) {
```

Particularly, BuCoR-E suggests nothing when no example exists or no exemplar edit is located in branches. For example, Listing 4.2 shows a conflict from swagger-core [33]. The conflict is of type C15—*Method: rename def vs. add use*, where r renames method *setRef(...)* and l adds an invocation to that method. Although an intuitive resolution is to update the newly added method call, no edit example in r demonstrates such change. On the other hand, we applied GumTree to compare different versions of source code, and derive AST edit scripts to represent code changes. As GumTree cannot detect any changes to `PackageDeclaration` or `ImportDeclaration`, BuCoR-E cannot extract such edits to suggest related resolutions.

BuCoR-R is applicable to many cases where (1) BuCoR-E does not work and (2) the needed resolutions are conventional, such as the case in Listing 4.2. However, for conflicts which have no conventional resolution method recommended by literature (see C17–C23 in Table 4.2), BuCoR-R generates nothing because there is no typical, generally accepted way to handle them. For example, Listing 4.3 shows a conflict from Jedis [35]. This conflict is of C20–*Field: remove def vs. add use*, where l removes field `timeout` and r introduces a new reference to `timeout`. Such a conflict may get resolved by adding back the field *def*, replacing the added *use* with something else, or simply removing that *use*; nevertheless, there is no standard

solution or typical resolution pattern followed by developers or documented in literature. Therefore, BuCoR-R did not handle it, while BuCoR-E resolved it in a project-specific way by referring to edit examples. It removed field *use* as below:

```
return new JedisClusterCommand<Set<String>>(connectionHandler, maxRedirections) {...}
```

Among the 21 conflict categories in our dataset, BuCoR resolved conflicts of 19 categories. It did not resolve any conflict of C19 and C22, as (1) BuCoR-E did not find any exemplar edit, and (2) there is no well-accepted resolution pattern for BuCoR to implement.

Finding 1 (Answer to RQ1): BuCoR *generated resolution(s) for 65 of the given 88 conflicts, achieving 74% coverage. Among the 79 resolutions it produced, 28 and 51 were separately contributed by* BuCoR-E *and* BuCoR-R.

4.4.4 RQ2: BuCoR Resolution Accuracy

As shown in Table 4.3, 21 of the 28 resolutions produced by BuCoR-E are correct, leading to 75% accuracy; 20 of the 51 resolutions output by BuCoR-R are correct, achieving 39% accuracy. Overall, BuCoR obtained 52% accuracy, having 41 of the 79 generated resolutions to be correct. These measurements also show that the two strategies complement each other. Among the 14 cases where both strategies suggest resolutions, BuCoR-E and BuCoR-R correctly resolved 13 and 7 cases, respectively; the correct resolutions overlap in 7 cases. Thus, BuCoR resolved 34 cases correctly.

Particularly, BuCoR-E resolved build conflicts by mimicking developers' project-specific solutions to related build errors. However, BuCoR-E did not always output correct solutions, because the expected ground-truth resolutions of some conflicts partially overlap with fixes to build errors, or present patterns totally different from those fixes. For instance, in Listing 4.4, l moves method `getSerializationService()` from class `ClientMap` to `ClientContext`,

while r adds two calls to the original method. BuCoR-E replaces the first method call with `getContext().getSerializationService()`, by following a mined example. Although this replacement is correct, BuCoR-E does not replace the second method call as expected since the original example only contains and updates one method call.

In another scenario (see Listing 4.5), r updates a constructor's signature by inserting a parameter of type `ExecutorService`, and l adds a call to the original constructor. BuCoR-E derived a resolution from branch edits, by updating the call to take an extra argument:

```
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors() * 2).
```

Although this edit can resolve the conflict, developers' manual resolution removes the constructor entirely. In such scenarios, branch edits do not reflect developers' preference of conflict resolution.

Listing 4.6: A conflict for which the well-accepted resolution pattern does not match developers' manual resolution [34]

```
1 /* Left Version: TypeUtils.java removes an import */
2 - import java.beans.Introspector;
3
4 /* Right Version: TypeUtils.java adds a reference to the imported class */
5 + if (compatibleWithJavaBean){
6 +  propertyName= Introspector.decapitalize(methodName.substring(3));
```

BuCoR-R sometimes suggested incorrect resolutions, because the conventional resolution patterns it incorporates do not always match developers' practices. For instance, the conflict in Listing 4.6 has l remove an import and r add a reference to the removed imported class. BuCoR-R resolved this conflict by adding back the removed import. However, developers implemented a replacement method, to avoid the removed dependency. Additionally, for conflicts due to class-hierarchy changes, BuCoR-R modified subclasses to match changes in

super classes, while developers sometimes adapted super classes to subclasses.

The 34 conflicts correctly resolved by BuCoR fall into 12 categories, implying that BuCoR can properly resolve diverse conflicts. There are seven categories of conflicts for which BuCoR produced some resolutions but none of them are correct: C2–C4, C8–C9, C12, and C18. Six of the categories (C2–C4, C8–C9, and C12) are about mismatches between super types and sub types; three of the categories are caused by parameter list changes (C3, C9 and C18). Among the seven incorrect resolutions suggested by BuCoR-E, two are partially incorrect, by presenting subsets of the needed changes; five are totally irrelevant. Among the 31 incorrect resolutions output by BuCoR-R, 15 are partially incorrect, while 16 are irrelevant.

Finding 2 (Answer to RQ2): BuCoR *generated in total 41 correct resolution(s), achieving 52% (41/79) accuracy.* BuCoR-E *and* BuCoR-R *separately contributed 21 and 20 correct resolutions.*

4.4.5 RQ3: BuCoR-E vs. BuCoR-R

BuCoR-E and BuCoR-R are different in two aspects:

First, BuCoR-R *has higher coverage (58% vs. 32%) by resolving more conflicts;* BuCoR-E *achieves higher accuracy by having more of its resolutions match developers' intents (75% vs. 39%).* It implies that if developers (1) want to automatically resolve as many conflicts as possible in conventional ways, and (2) have good testing to detect wrong resolutions, they can rely more on BuCoR-R. If developers (1) want to automatically resolve conflicts in unconventional ways but (2) have poor support to detect wrong resolution, they can rely more on BuCoR-E. For instance, when both strategies output resolutions (see Figure 4.4), it is likely that BuCoR-E's result is better.

Second, the two strategies are good at resolving different types of conflicts. Among the 34

conflicts correctly handled by BuCoR, 7 conflicts are correctly resolved by both strategies. In addition to that, BuCoR-E correctly resolved 14 cases; for 8 of these cases, BuCoR-R could not suggest anything. The eight cases are related to entity removal (i.e., C17, C20, C23) and parameter-list changes (i.e., C21). It means that BuCoR-E is better at resolving conflicts in unconventional ways. Meanwhile, BuCoR-R correctly resolved 13 cases, for which BuCoR-E suggested nothing. These 13 conflicts are mainly related to super-sub mismatches (C10–C11) and duplicated entities (C14, C16). It means that BuCoR-R is better at handling conflicts in conventional ways.

4.4.6 Runtime Overhead

On our dataset, BuCoR spent 0.1–53.6 minutes on each of the 55 merging scenarios, with 2.9 minutes as the mean and 0.8 minutes as the median. Among the four components of BuCoR-E, conflict detection and graph construction are the most time-consuming, roughly taking up 55% and 36% of the overall runtime. The time BuCoR spent on each merging scenario is closely related to the number of (1) Java files and (2) program entities. Namely, the more files to parse and the more entities to analyze, the longer BuCoR takes. We noticed that BuCoR spent the least time 0.1 minutes, when there are 20 Java files and 1,229 program entities under processing. It spent the most time 53.6 minutes when there are 1,472 Java files and 48,299 entities being analyzed.

4.5 Threats to Validity

Threats to External Validity. Our evaluation dataset consists of 88 real-world build conflicts, spanning 21 distinct conflict types. However, the program contexts and conflict patterns

covered by this dataset may not fully capture the diversity of build conflicts in real world. BuCoR-R includes 16 well-defined rules of handling frequently occurring conflicts. Although they cover all the typical conflict-resolution strategies mentioned by prior work [55, 91], they may not include all the frequently applied strategies in reality. In the future, to enhance our research representativeness, we will (1) expand our dataset to include more merging scenarios, and (2) enlarge the ruleset of BuCoR-R whenever possible.

Threats to Internal Validity. BuCoR-E leverages the threshold 0.618 to decide whether two AST nodes are similar enough to have a value match, because prior work [89] shows that this setting led to reasonably good results. However, there can be scenarios where two matching AST nodes have a lower similarity score than 0.618, disabling BuCoR-E to apply edits as expected. In the future, we will explore better ways of node matching.

Threats to Construct Validity. BuCoR leverages Bucond to detect conflicts. If Bucond cannot detect certain build conflicts (e.g., conflicts in build scripts), BuCoR cannot resolve those conflicts either. Prior work [99] shows that Bucond has a very good coverage of conflict types, and there is a low chance of Bucond to miss real-world conflicts in Java source code. Therefore, the effectiveness of BuCoR is not considerably limited by Bucond.

4.6 Summary

This paper introduces BuCoR, a novel conflict resolver to conduct example-based and rule-based transformation. It applies program static analysis to (1) detect conflicts, (2) mine edit examples in branches, (3) derive transformation patterns from mined examples, (4) and apply inferred or predefined patterns to resolve conflicts. Compared with prior work, it significantly pushes the boundary of automatic conflict resolution. Our investigation is deeper as (a) the conflicts we focus on are very diverse; (b) the edits we suggest vary a

lot depending on the program context and conflict types; (c) in the scenarios where no standard resolution pattern exists, BuCoR can create resolutions by locating, refining, and reusing relevant edits. In the future, we will improve BuCoR by (i) expanding the rule set of BuCoR-R, and (ii) improving the example mining capability of BuCoR-E when no local example is extractable from branches.

Chapter 5

How Effectively Do Large Language Models Help with Build Conflict Resolution?

5.1 Introduction

In collaborative software development, developers commonly employ branching workflows to support parallel feature implementation and bug fixing [12]. When branches are merged, edits from different contributors may interfere, producing inconsistencies that must be resolved during integration. These inconsistencies, known as *merge conflicts*, are among the most frequent and time-consuming integration challenges in collaborative development [58, 79]. Specifically, Brindescu et al. [48] analyzed 143 open-source projects, reporting that (1) almost 1 in 5 merges cause conflicts; (2) in 75% of these cases, a developer needed to reflect on the program logic to resolve conflicts. Nelson et al. [79] reported that developers defer responding to conflicts based on their perception of the complexity of the conflicting code, and that deferring affects the workflow of the entire team.

As shown in Fig. 5.1, a typical merging scenario involves five program versions. Two **branch versions**—l and r—contain the independent edits to be merged. The **base version (b)**

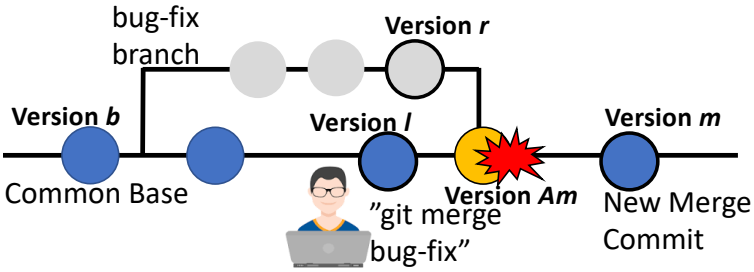

Figure 5.1: A merging scenario may involve five program versions

Changes in local (*l*) branch	Changes in remote (*r*) branch

public class B {
public void bar() {
+ A.m();
... } ... }
public class A {
public static void m() { ...
} }
	public class B {
	public void bar() {
	...
	} ... }
	public class A {
	- public static void m() { ... }
	+ public static void m2() {...}
	}

Figure 5.2: A build conflict related to method renaming

represents their common ancestor. The **automatically merged version (A_m)** is produced by merge tools such as git-merge, which attempt to combine branch edits textually. The **manually merged version (m)** is the corrected result developers produce after resolving conflicts in A_m.

Merge conflicts can be broadly divided into three categories. **Textual conflicts** arise when branch edits modify the same line(s) of code in divergent ways. **Build conflicts** occur when incompatible edits lead to build or compilation errors in the merged version. As illustrated in Fig. 5.2, for example, because l adds a call to `m()` while r renames that method, the co-application of both edits results in an unresolved method reference. Namely, the newly added method call `m()` does not correspond to any defined method, thereby causing a compilation error. **Test conflicts** emerge when a merged version compiles successfully but fails one or more tests, exhibiting runtime behaviors inconsistent with the intended program semantics.

Our research focuses on **resolving build conflicts**, as such conflicts often introduce subtle and hard-to-diagnose compilation errors that can significantly delay software integration and deployment [90, 97].

People created merge tools to tentatively integrate software branches, reveal conflicts in the process, or resolve conflicts with the best effort. Nevertheless, these tools are limited in various ways. First, textual or unstructured merge tools (e.g., git-merge [39] and KDiff3 [57]) combine branch edits textually, but detect textual conflicts with lots of false positives [92]. Second, structured and semi-structured merged tools (e.g., JDime [42], FSTMerge [44], AutoMerge [111]) extract and compare abstract syntax trees (ASTs), to detect textual conflicts with higher rigor. However, none of the tools mentioned above handle build conflicts. Third, Bucond [99] and the approach of Wuensche et al. [101] statically analyze branch versions, to detect build conflicts based on pattern matching. However, neither tool resolves build conflicts.

Fourth, MrgBldBrkFixer [97] and GMerge [108] resolve build conflicts by updating newly added *use*s of entities (i.e., classes and methods), which entities have *def*s simultaneously renamed. See Fig. 5.2 as an example. To resolve the conflict, both tools would suggest updating the added call `A.m()` to `A.m2()`, to adapt to the method-renaming change in class `A`. However, these tools have limited application scope, as conflict resolution is not always as trivial as identifier renaming [90].

To better help developers resolve build conflicts, this paper introduces our systematic study that investigates using large language models (LLMs) to resolve build conflicts. We hypothesized that **given sufficient program context of branches-to-merge and/or domain knowledge of frequently adopted conflict resolution strategies, LLMs are capable of resolving build conflicts that fit both the program context and conflicting scenarios.** To verify our hypothesis, we designed six prompt templates (P1–P6) to cover four

information elements: (1) minimized difference (mini diff) to present the conflict location and minimal context, (2) full diff as the most comprehensive context of a conflicting scenario, (3) resolution rules as the domain knowledge of frequently applied resolution strategies, and (4) exemplar edits to show candidate resolution-relevant edits extractable from branches.

In our study, we used the dataset of build conflicts from prior work [99]. We queried LLMs to resolve those conflicts by sending prompts generated with the aforementioned templates to state-of-the-art LLMs: Claude Sonnet 4 [28], Gemini 2.5 Flash [27], Llama 4 Maverick [37], and OpenAI o4-mini [36]. We manually compared the output by different models against developers' resolutions, to decide whether the suggested resolutions are correct. Based on the model outputs and our manual analysis, we explored the following research questions (RQs):

- **RQ1: Which prompt design is the most effective?** Among the six templates, P3 (mini diff + full diff) is the most effective, enabling Flash to correctly resolve 59 conflicts. P2 (full diff) is the least effective: o4-mini only resolved 24 conflicts given such prompts.
- **RQ2: How do different elements of prompts help with LLM-based conflict resolution?** Mini diff helps precisely locate conflicts. Resolution rules are less helpful; they sometimes even mislead LLM-based approaches. Exemplar edits help only when they are extracted from the same merging scenario as the conflict-to-resolve.
- **RQ3: How do different LLMs balance between the resolution effectiveness and their cost?** Expensive models do not necessarily outperform. Given P3 prompts, the second cheapest model Flash achieved the best result, significantly outperforming the most costly model Sonnet-4.

To sum up, we made the following research contributions:

Edits from the branches-to-merge

(a) Conflicting edits between branches

Changes in ***l*** (responsible for def removal):
In TypeUtils.java, an import declaration is removed.

```
- import java.beans.Introspector;
  ...
```

Changes in ***r*** (responsible for use addition):
In the same Java file, code is updated to use the class.

```
  … } else if (c3 == 'f') {
     propertyName = methodName.substring(3);
+    } else if (methodName.length()>=5 &&
Character.isUpperCase(methodName.charAt(4))){
+      propertyName =
Introspector.decapitalize(methodName.substring(3));
     } …
```

(b) An exemplar edit E extracted from a branch to adapt code

Edit example in *l* to adapt usage of `Introspector`:
Replace the method call with a different one

```
  …
- propertyName =
Introspector.decapitalize(methodName.substring(3));
+ propertyName = decapitalize(methodName.substring(3));
  …
- propertyName =
Introspector.decapitalize(methodName.substring(2));
+ propertyName = decapitalize(methodName.substring(2));
  …
```

Edits applicable to A_m, to resolve conflicts

(c) A naïve resolution producible by conventional rules, without aligning with developers' intent

Re-add the removed import

```
+ import java.beans.Introspector;
```

(d) A project-specific resolution that aligns better with developers' intent

Replace the method call with a different one

```
- propertyName =
Introspector.decapitalize(methodName.substring(3));
+ propertyName = decapitalize(methodName.substring(3));
```

Figure 5.3: A motivating example of build conflict, which cannot get handled by existing tools or conventional rules

1. **Empirical study:** We systematically explored LLM usage for build conflict resolution in Java programs. No prior work studied this in such a comprehensive way.

2. **Prompt taxonomy:** We designed and adopted six structured prompt templates, to query LLMs with diverse information elements. No prior work studied any of these prompts or elements.

3. **Novel results:** We revealed important findings for the first time, on LLMs' capability of conflict resolution and the effectiveness comparison between different prompts.

4. **Model comparison:** We compared four state-of-the-art LLMs in terms of their resolution capability and cost, to provide insights on future model selection or usage.

5.2 A Motivating Example

To facilitate discussion, this section introduces a running example from a real-world project fastjson [34]. As shown in Fig. 5.3 (a), a merging scenario has l remove a class import and r insert code to use that imported class. The naïve integration of these edits can trigger a build error because the newly added *use* refers to a nonexistent class *def*. Manually detecting and resolving build conflicts can be tedious and error-prone. No existing tool resolves such conflicts.

To automatically resolve the aforementioned conflict, a naïve approach may add back the deleted import (Fig. 5.3 (c)), by following a conventional strategy mentioned in prior work [55]. Although this "import re-addition" appears to fix the compilation error, it fails to respect developers' actual change intent. As shown in Fig. 5.3 (b), while developrs removed the import in branch l, they also replaced all calls to `Introspector.decapitalize(...)` with calls to a locally defined method `decapitalize(...)`, to eliminate dependencies on the removed class import. Consequently, the developers' intended resolution, as depicted in Fig. 5.3 (d), was to update the newly added method call.

This motivating example highlights that successful automatic build-conflict resolution requires not only domain knowledge of diverse resolution strategies, but also an ability to infer developers' change intent from branch edits and apply context-aware fixes. Unfortunately, existing build-conflict resolvers (i.e., MrgBldBrkFixer [101] and GMerge [108]) offer only limited support—they lack both a comprehensive rule base and the capability to infer developer intent.

Recently, large language models (LLMs) have demonstrated remarkable abilities to reason across the syntactic, semantic, and contextual aspects of source code [41, 54, 71, 72, 73, 76, 103, 104, 104, 105, 110]. Motivated by the extensive domain knowledge encoded in LLMs

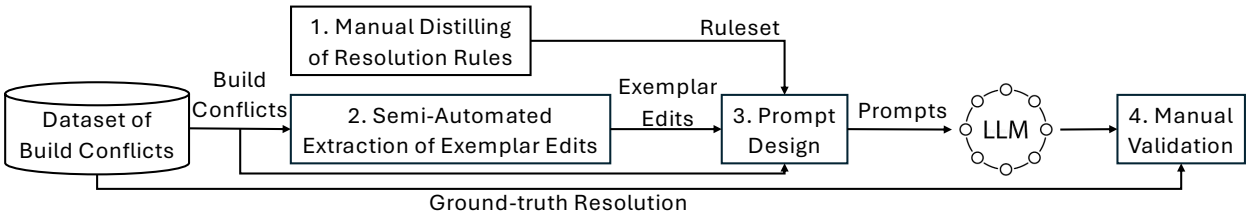

Figure 5.4: Our study methodology consists of four phases

and their strong reasoning capabilities, in this paper, we investigate to create and assess LLM-based build-conflict resolvers that aims to overcome the key limitations of existing tools.

5.3 Study Methodology

Our methodology comprises four phases (see Fig. 5.4). The first two phases focus on constructing **knowledge bases** that serve as inputs for querying LLMs. Specifically, Phase I builds a knowledge base of *conventional build-conflict resolution rules* through a manual curation process. Phase II begins with a third-party dataset of build conflicts [99], and semi-automatically derives a knowledge base of *exemplar edits* that may assist in resolving similar conflicts. Afterwards, Phase III integrates build-conflict cases with the knowledge bases produced in Phases I and II to query LLMs, using six alternative prompt templates. Finally, Phase IV compares LLM-generated resolutions against the ground-truth fixes, to assess the effectiveness of conflict resolution. Sections 5.3.1–5.3.4 explains each phase in detail.

Table 5.1: The 16 resolution rules we manually mined from existing studies [55, 90, 97]

Idx	**Conflict Type**	**Resolution Pattern**
C1	Class: rename def vs. add use	Update the added use
C2	Class: add method def in super vs. add sub class	Update method def in sub to match super
C3	Class: change a method's parameter list in super vs. add sub class	Update method def in sub to match super
C4	Class: change a method's return type in super vs. add sub class	Update method def in sub to match super
C5	Import: remove def vs. add use	Re-add the def (i.e., add back the removed entity import)
C6	Package: rename def vs. add use	Update the added use (i.e., update import with the new package name)
C7	Interface: rename def vs. add use	Update the added use
C8	Interface: add method def in super vs. add class to implement the super	Add/update method def in class to override the new method in super
C9	Interface: change a method's parameter list in super vs. add class to implement the super	Update parameter list in class to match super
C10	Interface: remove method def in super vs. add class to implement the super	Remove method def in class to match super
C11	Interface: rename a method def in super vs. add class to implement the super	Rename method def in class to match super
C12	Interface: change a class to implement the interface vs. change a method's return type in the class	Update return type of method in class to match super
C13	Field: rename def vs. add use	Update the added use
C14	Field: add def vs. add def	Remove redundant field definition
C15	Method: rename def vs. add use	Update the added use
C16	Method: add def vs. add def	Remove redundant method definition

5.3.1 Phase I: Manual Distilling of Resolution Rules

Researchers conducted studies to characterize build conflicts and their resolution strategies [55, 97]. To extract conventional (i.e., frequently applied) resolution strategies, we systematically reviewed these studies and compiled a list of eight rules. A rule was included into this list if it was the most frequently adopted strategy for a given type of scenarios, or was illustrated with clear representative example(s). By generalizing rules across similar conflict types (e.g., from method renaming-related conflicts to field renaming-related conflicts), we derived 16 rules as the knowledge base of frequently adopted resolution strategies. As shown in Table 5.1, C1, C3, C5-C6, C8, and C14-C16 are the eight rules we observed in prior work; the remaining rules were generalized from those observed rules.

Among the 16 conflict types, 14 stem from mismatches between the definition (i.e., *def*) of program entities (i.e., packages, imports, classes, interfaces, methods, and field) and their usage (i.e., *use*); the remaining 2 arise from redundant entity definitions. To resolve conflicts due to def-use mismatches, developers typically adapt entity *uses* to *defs*. For instance, C1 captures cases where one branch renames a class *def* while the other adds a new *use* of that class; developers commonly address such conflicts by updating the added *use* to match the renamed *def*. Similar principles underlie rules C2–C4, C6–C13, and C15. C5 is an exception, as it adapts entity *def* to *use(s)*, probably because editing import declarations is simpler than editing other code. Finally, conflicts arising from duplicated *defs* are generally resolved by de-duplicating, as shown by C14 and C16.

5.3.2 Phase II: Semi-Automated Extraction of Exemplar Edits

Prior studies [55, 97] show that many build conflicts arise from def-use mismatches, which were introduced when semantically inconsistent edits from different branches get integrated.

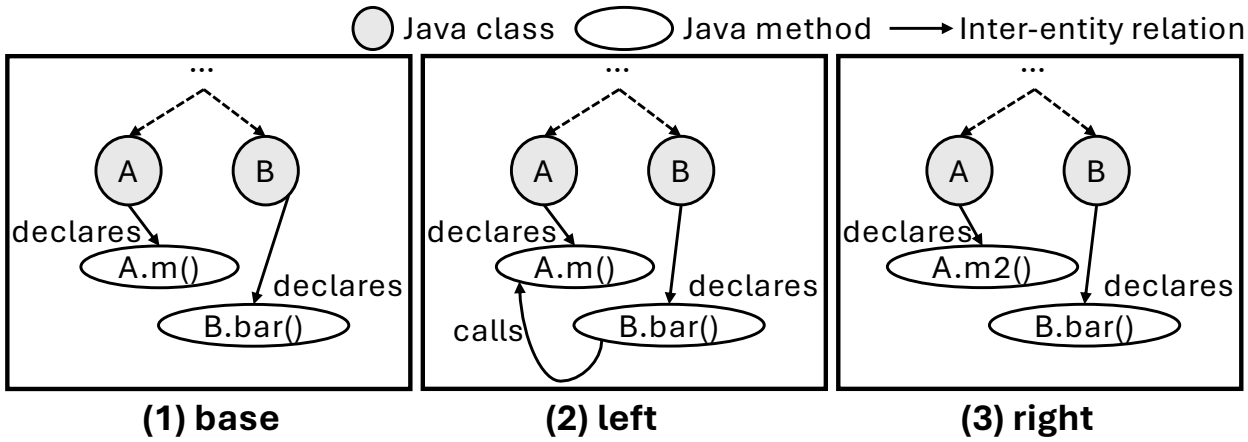

Figure 5.5: The PEGs Bucond [99] created for the running example

Similarly, many build errors within the a single branch result from def-use mismatches involving the same program entities. This observation suggests a commonality between **how developers resolve build conflicts** and **how they fix build errors in individual branches**. Thus, we hypothesize that *if developers have previously addressed def-use mismatches on a branch through specialized edits, they are likely to apply similar edits when resolving the same kind of mismatches during a software merge.* Guided by this insight, we designed a semi-automated approach to extract exemplar edits from branch versions, for knowledge-base construction.

Our semi-automated extraction process consists of two steps: (1) Bucond-based extraction of entity-level edits, and (2) example localization through a combination of automated and manual analysis. Bucond [99] is a detector of build conflicts based on static analysis. As illustrated in Fig. 5.5, given a merging scenario depicted with three program versions (b, l, and r), Bucond models each version as a **program entity graph (PEG)**, to capture defined program entities (e.g., classes and methods) and their relations (e.g., containment or method invocations). By comparing PEGs, Bucond identifies entity-level edits in l and r. It then matches these edits against 57 predefined patterns of conflicting edits. For instance, one pattern checks when a method is renamed in one branch, whether the other branch adds

any call to the original method; if so, Bucond reports a conflict.

Step 1 extends Bucond to create PEGs for a given build conflict, and to locate the *def*-change responsible for the reported conflict. Step 2 navigates PEGs, to find *use*-changes connected to the responsible *def*-change in the same branch version. If a single edited entity E is found to adapt *uses* to the *def*-change, the before- and after-versions of E are retrieved as an **exemplar edit**. If multiple entities are edited to adapt *uses*, we randomly select one as the exemplar. If no edited entity is located, it means that developers did not adapt any *use* to the *def*-change, and no exemplar edit is extractable.

5.3.3 Phase III: Prompt Design

To deepen our investigation, we designed six prompt templates. Each prompt follows a consistent three-part structure:

(1) **Definition (D)**—a concise description of build conflicts;

(2) **Inputs (I)**—details of the conflicting merge scenario along with optional knowledge bases constructed in Phases I–II, and requests for resolution suggestion;

(3) **Output Instructions (O)**—specifications of the expected format of the model's response.

While all six templates share parts (1) and (3), they differ in part (2), by providing varying degrees of code context and domain knowledge to assess the sensitivity of LLM performance to input richness.

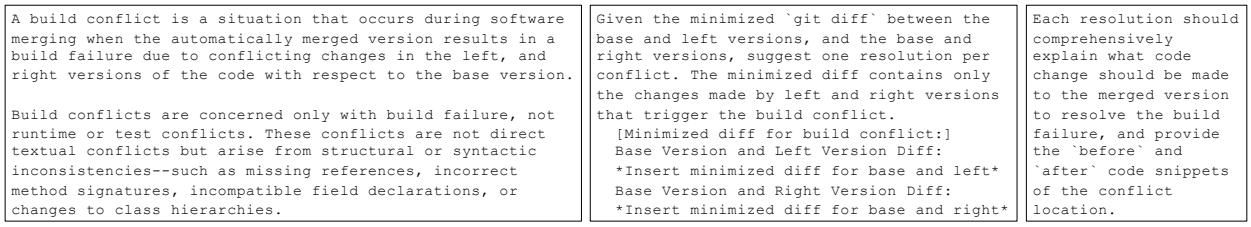

Figure 5.6: Exemplar fragments of prompt templates

Definition (D)

All prompts begin with the text shown in Fig. 5.6(a) to clarify (1) what a build conflict is, (2) how it differs from textual and test conflicts, and (3) some possible causes of build conflicts.

Inputs (I)

The second part of each prompt introduces the branch edits and optional knowledge bases. They characterize the six alternative templates in the following ways.

P1. Minimized Difference (Mini Diff) provides the minimized `git diff` output between b and l, and between b and r. These diffs capture only the essential edits directly responsible for a reported build conflict. When supplied with such prompts, LLMs are expected to identify conflicts in the limited context, and resolve those conflicts. For our motivating example, P1 uses the content shown in Fig. 5.3(a) to instantiate the template fragment shown in Fig. 5.6(b).

P2. Full Diff provides the complete `git diff` between the base and each branch version, to expose the full context of branch edits. Intuitively, these prompts are harder to handle than P1 prompts, as the expanded context introduces (1) data noise—code unrelated to the actual conflicts or their resolutions, and (2) additional branch edits to enlighten project-

specific context-aware resolutions.

P3. Mini Diff + Full Diff combines the `git diff` outputs mentioned in P1 and P2, providing LLMs with access to both local and global change contexts. This design enables LLMs (1) to reason about the relationships between localized conflicting edits and the broader code surroundings, and (2) to resolve the identified conflicts using the immediate context and additional branch edits.

P4. Mini Diff + Resolution Rules augments minimized diffs with the knowledge base of 16 resolution rules constructed in Phase I. This design intends to equip LLMs with explicit domain knowledge of commonly applied conflict-resolution strategies, which may not be fully captured in the pretraining of LLMs. The template presents each rule following a consistent structure, specifying (1) the conflict type, (2) a summary of left-branch edits, (3) a summary of right-branch edits, and (4) the corresponding resolution strategy. For example, the rule associated with Fig. 5.2 is presented in the template as follows:

```
Rule 15:
- Pattern: Method: rename def vs. add use
- Left Version: Rename method
- Right Version: Add reference to method
- Resolution Strategy: Update reference with new method name
```

P5. Mini Diff + Exemplar Edits augments minimized diffs with the knowledge base of context-aware exemplar edits generated in Phase II. This design requires LLMs to carefully examine the provided examples, and to strictly follow the exemplar edits whenever applicable. Our goal is to expose LLMs to concrete branch edits that address build errors similar to those triggered by the current build conflict. In this way, even if an LLM is unable to in-

dependently locate relevant examples within branch versions, the supplementary knowledge can effectively compensate for that limitation. The template represents each edit example in the knowledge base with a uniform structure, specifying (1) the conflict type, (2) fully qualified name of the conflict-relevant program entity, (3) the Java file before the edit, and (4) the file after the edit.

P6. Mini Diff + Resolution Rules + Exemplar Edits augments mini diff with both knowledge bases. This design aims to fuse domain knowledge from both sources, to offer complementary guidance that may help LLMs better resolve conflicts. Specifically, when a merging scenario does not have any branch edits similar to the needed resolution edits, resolution rules can assist LLMs in proposing meaningful, generalizable fixes. Conversely, if a scenario has some branch edits to imply the required resolution, exemplar edits enable LLMs to prioritize project-specific resolutions over broader, project-agnostic ones.

Output Instructions (O)

All prompts conclude with the standard output specification shown in Fig 5.6(c). It requests (1) a rationale behind each suggested resolution, and (2) the before- and after- versions of each edited location. This standard requirement ensures consistency in the structure and content of LLM outputs, enabling a fair and reliable evaluation across different prompt settings.

5.3.4 Phase IV: Manual Validation

After executing LLMs with different prompts, we manually inspected the models' outputs to interpret all detected conflicts and resolution suggestions. We compared that information with ground-truth for correctness labeling. To ensure the reliability of this procedure, three

Table 5.2: Measurement comparison between different prompt designs

Metric	P1 (Mini diff)	P2 (Full diff)	P3 (Mini + Full diff)	P4 (Mini + Rules)	P5 (Mini + Examples)	P6 (Mini + Examples + Rules)
A	100% (83/83)	92% (76/83)	92% (76/83)	100% (83/83)	100% (83/83)	100% (83/83)
P	98% (83/85)	50% (62/123)	95% (76/80)	98% (83/85)	100% (80/80)	100% (83/83)
R	98% (81/83)	75% (62/83)	92% (76/83)	100% (83/83)	96% (80/83)	100% (83/83)
F	98%	60%	93%	99%	98%	100%
C	48% (40/84)	39% (24/62)	68% (51/76)	47% (39/83)	58% (46/80)	55% (46/83)

authors separately reviewed and labeled all LLM outputs. They then compared all labels and discussed any discrepancies until a consensus was reached.

5.4 Experiments

With the LLM-based conflict resolution approaches described in Section 5.3, we conducted experiments to explore the following three research questions (RQs):

- **RQ1:** *Which prompt design is the most effective?*
- **RQ2:** *How do different information elements of prompts help with LLM-based conflict resolution?*
- **RQ3:** *How do different LLMs balance between the resolution effectiveness and their cost?*

In the remaining part of this section, we will first introduce our experiment settings (Section 5.4.1) and the evaluation metrics (Section 5.4.2). Next, we will introduce and explain the experiment results (Section 5.4.3).

5.4.1 Experiment Settings

To conduct experiments, we reused an open-source dataset of build conflicts from the Bucond paper [99] for two key reasons. First, the dataset consists of 55 real-world merging scenarios, corresponding to 83 build conflicts. Specifically, 81 of the conflicts were labeled by the original ground-truth; the remaining 2 conflicts were revealed by Bucond and confirmed but not documented by the original dataset. These conflicts encompass 21 distinct conflict types. This diversity enables a comprehensive evaluation of LLM-based conflict resolution. Second, in each merging scenario, the four program versions—b, l, r, and m—are all compilable, whereas the naïvely merged version A_m is not. This property ensures that the branch edits are of high quality, and any compilation errors in A_m can be attributed solely to build conflicts. Moreover, since the developer-generated versions m are always compilable, they can serve as reliable ground-truth references for us to assess tool-generated resolutions.

In Phase II of our methodology (Section 5.3), we could only extract 22 examples from branch versions in this dataset, which examples try to resolve similar build errors as those triggered by build conflicts in the automatically merged version A_m. We integrated these examples into the prompts generated by P5 and P6.

By default, we used OpenAI o4-mini [36] to execute all six prompt templates, as it is one of the state-of-the-art models at the time of experimentation. While our methodology is not tailored to any specific model, we acknowledge that different LLMs may exhibit varying performance under the same prompt conditions. After evaluating o4-mini's results across the six prompt templates, we selected the template yielding the highest effectiveness as the default configuration for our approach. We then experimented with three additional LLMs—Claude Sonnet 4 [28], Gemini 2.5 Flash [27], and Llama 4 Maverick [37]—to compare their effectiveness and cost-efficiency. All models were accessed through public APIs between

August–December 2025.

5.4.2 Evaluation Metrics

We used the following five metrics to evaluate LLM-based build conflict resolution:

Tool Applicability (A) measures the proportion of build conflicts for which an LLM-based conflict resolver can process the inputs and produce conflict-relevant information.

$$A = \frac{\text{\# of conflicts processable}}{\text{Total \# of build conflicts}} \tag{5.1}$$

Given a conflict, an LLM is inapplicable if it cannot process the input (i.e., the input exceeds the LLM's maximum token limit), or it outputs information totally irrelevant to the conflict.

Detection Precision (P) measures the proportion of conflicts reported by an approach that are actual, valid conflicts.

$$P = \frac{\text{\# of conflicts correctly reported}}{\text{Total \# of reported conflicts}} \tag{5.2}$$

Detection Recall (R) measures the proportion of known conflicts (i.e., the ground-truth set) reported by an approach.

$$R = \frac{\text{\# of conflicts correctly reported}}{\text{Total \# of known conflicts}} \tag{5.3}$$

Detection F-score (F) is the harmonic mean of precision and recall, to reflect the overall effectiveness of an approach's conflict detection.

$$F = \frac{2 \times P \times R}{P + R} \tag{5.4}$$

Resolution Accuracy (C) measures among all suggested resolutions, the proportion that are correct.

$$C = \frac{\text{\# of resolutions that are correct}}{\text{Total \# of suggested resolutions}} \tag{5.5}$$

A resolution is correct if the suggested edits are *semantically equivalent* to the ground-truth edits—that is, the proposed resolution achieves the same effect as developers' actual fix.

All metrics range from 0 to 1, with higher values indicating better performance. Given a conflict, if a tool is inapplicable, we do not evaluate the remaining metrics. Likewise, if a tool incorrectly reports a build conflict, we do not assess the correctness of any resolution it proposes for that conflict.

5.4.3 Results

This section presents and explains our experiment results for each RQ in detail.

Effectiveness Comparison between Prompt Designs (RQ1)

As illustrated in Table 5.2, o4-mini achieved very diverse results when given differently generated prompts.

Applicability: The model is always applicable when given prompts of P1, and P4–P6. It is inapplicable in seven cases under P2–P3 prompts. This is because both P2 and P3 include the *full diff* of branch versions. When the full diff exceeds 200,000 token-limit of o4-mini, the model is unable to process the input or execute any task.

Conflict Detection: With P6 prompts, o4-mini achieved the best performance, detecting conflicts with 100% precision and 100% recall. The next-best performance was obtained using P1 and P4 prompts, followed by P5. In contrast, P2 prompts yielded the worst results: 50% precision and 75% recall. P3 prompts performed better than P2—achieving 95% precision and 92% recall, but still lagged behind P1 and P4–P6. Overall, the F-score comparison between templates is P6 > P4 = P1 > P5 > P3 > P2.

Three reasons help explain the observation. First, all templates except P2 include *mini diffs*, which effectively constrain the search scope of candidate conflicts, and highlight the regions containing conflicts. Such localized information is sufficient for o4-mini to reason about context and identify root causes. Without mini diffs, P2 left the model to reason solely from full diffs and caused substantially more errors, including both false positives and false negatives. Second, when both mini diff and full diff are available in P3 prompts, the additional program context introduces noise that can mislead the model. As a result, in two cases under P3, the LLM detected more conflicts than actually present. Third, o4-mini was unable to process seven large-input cases under P2 and P3, so the recall for these templates was further reduced.

Conflict Resolution: With P3 prompts, o4-mini obtained the highest resolution accuracy: 67%. P5 and P6 performed similarly, with accuracies of 58% and 55%, respectively. P1 and P4 also exhibited comparable performance (48% and 47%). P2 prompts yielded the lowest accuracy at 39%. In terms of absolute numbers, P3 resolved the most conflicts (51); P5 and P6 each resolved 46; and P2 resolved the fewest—24. In short, the resolution accuracy comparison is P3 > P5 > P6 > P1 > P4 > P2.

Among the 83 conflicts, 15 conflicts were always resolved by o4-mini, no matter which template was used. These conflicts belong to four major categories. First, when one branch renames an entity and the other introduces a new *use* of that entity, the resolution is to

updates the *use* to reference the new name. Second, when one branch updates a method's parameter list by adding a parameter or updating a parameter type, while the other adds a new call to that method, the resolution is to update that call accordingly. Third, when one branch removes a class import while the other introduces a new *use* of the class, the resolution is to add back that import. Fourth, when both branches add the same entity, the resolution is to remove one of the duplicates. These consistently successful cases suggest that o4-mini possesses strong pretrained domain knowledge of frequently adopted resolution rules.

O4-mini never resolved 17 of the 83 conflicts, regardless of the template used. Two factors help explain the model's consistent failures in these cases. First, 12 conflicts require **project-specific** resolutions that deviate from commonly adopted practices, and the program context provides little or no cue on the expected solutions. Namely, when developers intentionally apply resolutions that contradict with the "*usual*" patterns, the model fails. For example, when a branch renames an entity or updates a method signature while the other adds a *use* of that entity/method, the ground-truth surprisingly removes the *use* while the model tries to adapts that *use*.

Second, six failures stem from the model's difficulty with fine-tuned, low-level implementation details. For instance, when a branch changes a method's parameter list or return-type in the base class while the other adds a class to extend that base, the correct resolution is to update the subclass method implementation to match the new signature. However, o4-mini fails to generate the necessary details, such as adjusting invocation parameters or adding required statements in the revised method, resulting in incomplete or incorrect patches.

Finding 1: *O4-mini achieved 100% applicability with prompts of P1 and P4–P6; it detected conflicts most effectively given P6 prompts, but resolved conflicts most effectively when using P3 prompts.*

Usefulness of Different Information Elements (RQ2)

With the six templates defined, we investigated the usefulness of four information elements and their combinations:

Mini Diff: As discussed in Section 5.4.3, *mini diffs* effectively constrain the search scope of candidate conflicts, enabling the LLM to detect conflicts with high accuracy. Therefore, mini diffs consistently play a positive role by helping with conflict detection, to further enable more reliable conflict resolution.

Full Diff: The impact of full diffs is mixed. On the one hand, the substantial data in full diffs disabled the application of o4-mini to seven cases under P2 and P3, reducing the model applicability. Additionally, with lots of data noise in P2 prompts, o4-mini did not work well to locate conflicts in full diffs. Thus, full diffs have a negative effect on tool applicability and conflict detection. On the other hand, with P3 prompts, o4-mini achieved the highest conflict-resolution effectiveness. When combined with the mini diff, the full diff provided complementary global context that the model was able to exploit. This combination enabled o4-mini to relate localized edits to the broader program structure, leverage resolution strategies implicitly learned during pretraining, identify relevant exemplar edits, and apply its domain knowledge effectively. Thus, while detrimental in isolation, the full diff plays a positive role in conflict resolution when used together with the mini diff.

Resolution Rules: When comparing o4-mini's effectiveness (1) between P5 and P6, and (2) between P1 and P4, we found that incorporating manually curated resolution rules tends to slightly (a) improve the recall of conflict detection, yet (b) worsen resolution accuracy.

Two factors likely explain such phenomena. First, compared with the broad and diverse

resolution strategies that o4-mini has implicitly learned during pretraining, our manually curated rule set is less comprehensive and may not fully represent the range of real-world resolution strategies. As a result, the rules sometimes help highlight conflicting patterns, but diverge from the resolution strategies actually needed in certain cases. Second, we found that the LLM occasionally overgeneralized some documented rules, inferred incorrect rules, and subsequently proposed invalid resolutions. As illustrated in Fig. 5.7, one conflicting scenario in Jedis [35] involves branch l removing a field `timeout` while branch r adds a method that uses `timeout`. The ground-truth resolution is to remove the field usage from the newly added method. However, based on our rule set, o4-mini overgeneralized the following import-related rule and incorrectly reintroduced the removed field:

```
Rule 5:
- Pattern: Import: remove def vs. add use
- Left Version: Remove import
- Right Version: Use the imported class
- Resolution Strategy: Re-add the def (i.e., add back the removed entity
import)
```

This misapplication demonstrates how manually curated rules can mislead the model. Overall, the resolution rules we provided play a negative role in conflict resolution.

Exemplar Edits: When comparing o4-mini's effectiveness (1) between P1 and P5 and (2) between P4 and P6, we found that providing manually located edit examples improves resolution accuracy. This is an interesting observation, as it suggests that when o4-mini is not given sufficient context-specific resolution-related edits, it may fail to suggest the correct resolution edits.

Changes in local (*l*) branch	**Changes in remote (*r*) branch**
private int timeout;	+ public Set<String> spop(final String key, final long count) { + return new JedisClusterCommand<Set<String>>(connectionHandler, **timeout**, maxRedirections) { … } + }

Figure 5.7: A build conflict where l removes a field and r adds a method to use that field [35]

(1) Changes in local (*l*) branch	**(2) Changes in remote (*r*) branch**
+ public RedisClient(String host, int port, int connectTimeout, int commandTimeout) { **+ this(new HashedWheelTimer(), new NioEventLoopGroup(), …, connectTimeout, commandTimeout);** + }	- public RedisClient(final Timer timer, EventLoopGroup group, …, + public RedisClient(final Timer timer, **ExecutorService executor,** EventLoopGroup group, …, int connectTimeout, int commandTimeout) {
(4)Developers' actual resolution edit	**(3) Exemplar edit extracted from *r***
Remove newly added constructor by *l*	- this(new HashedWheelTimer(), new NioEventLoopGroup(), NioSocketChannel.class, …, 10000, 10000); + this(new HashedWheelTimer(), **Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors() * 2)**, new NioEventLoopGroup(), …, 10000, 10000);

Figure 5.8: A build conflict where the exemplar edit extracted from r is not aligned with developers' actual resolution edit

Table 5.3: Comparison between different LLMs when P3 is used (Effectiveness Metrics)

LLM	**Effectiveness Metrics**				
	Applicability (A)	**Conflict Detection**			**Conflict Resolution**
		P	**R**	**F**	**C**
Claude Sonnet 4 [28]	99% (82/83)	100% (82/82)	99% (82/83)	99%	66% (54/82)
Gemini 2.5 Flash [27]	98% (81/83)	96% (79/82)	95% (79/83)	96%	75% (59/79)
Llama 4 Maverick [37]	76% (63/83)	91% (58/64)	70% (58/83)	79%	36% (21/58)
OpenAI o4-mini* [36]	92% (76/83)	95% (76/80)	92% (76/83)	93%	68% (51/76)

* By default, we used o4-mini to experiment with different prompt designs

Table 5.4: Comparison between different LLMs when P3 is used (Cost Metrics)

LLM	**Cost Metrics (total for the entire dataset)**		
	# of Input Tokens	**# of Output Tokens**	**Price in US Dollar**
Claude Sonnet 4 [28]	3,668,238	34,753	19.07
Gemini 2.5 Flash [27]		57,149	1.39
Llama 4 Maverick [37]		117,305	0.76
OpenAI o4-mini* [36]		29,447	2.40

* By default, we used o4-mini to experiment with different prompt designs

(1) Changes in local (*l*) branch	**(2) Changes in remote (*r*) branch**
In TypeSerializerConfig.java, names of the file, Java class, and constructor are all updated to SerializerConfig	Add a new class to use TypeSerializerConfig + ... + if ("**type-serializer**".equals(name)) { + **TypeSerializerConfig typeSerializerConfig** = new **TypeSerializerConfig**(); + **typeSerializerConfig**.setClassName(value); + ...
(3) Exemplar edit extracted from *l*	**(4) Developers' actual resolution edit to the added code**

- if ("**type-serializer**".equals(name)) {
- **TypeSerializerConfig typeSerializerConfig** = new **TypeSerializerConfig**();
- **typeSerializerConfig**.setClassName(value);
+ if ("**serializer**".equals(name)) {
+ **SerializerConfig serializerConfig** = new **SerializerConfig**();
+ **serializerConfig**.setClassName(value);
	- if ("**type-serializer**".equals(name)) {
	- **TypeSerializerConfig typeSerializerConfig** = new **TypeSerializerConfig**();
	- **typeSerializerConfig**.setClassName(value);
	+ if ("**serializer**".equals(name)) {
	+ **SerializerConfig serializerConfig** = new **SerializerConfig**();
	+ **serializerConfig**.setClassName(value);

Figure 5.9: A build conflict that was resolved by Flash, but not by any other model

Between P1 and P5, 6 additional conflicts were successfully resolved when P5 supplied 22 exemplar edits. These additional examples did not further enhance o4-mini's resolution accuracy under P1 (48%) for three main reasons. First, three of the exemplar edits were not aligned with the actual developer-applied resolutions; these misleading examples caused o4-mini to produce incorrect edits. As shown in Fig. 5.8, r updates a constructor to accept an extra `ExecutorService` parameter, whereas l adds a method to call the original constructor. To adapt all uses of the original constructor, r modifies each call site by adding a parameter `Executors.newFixedThreadPool(...)`, as illustrated by Fig. 5.8 (3). However, this systematic update was not adopted by developers when they resolved the build conflict after merging. Instead, developers removed the newly added constructor by l, making any edit analogous to the exemplar edit invalid.

Second, 10 of the conflicts already resolved under P1 required only straightforward renaming changes, which o4-mini can infer directly from the mini diff without any additional context. Consequently, although exemplar edits were provided to these cases, they did not yield further improvements in resolution accuracy. Third, exemplar edits were used exclusively to resolve conflicts within the same merging scenarios; they were never used to resolve conflicts from different scenarios with distinct program structures or entity usage.

Similarly, between P4 and P6, 7 additional conflicts got resolved successfully when P6 supplied 22 exemplar edits.

Finding 2: *Among the four elements, mini diff and exemplar edits help improve conflict resolution. Surprisingly, resolution rules do not enhance accuracy and may even degrade performance. Full diff, while detrimental when used in isolation, significantly boosts resolution accuracy when combined with the mini diff.*

Cost-Effectiveness Trade-offs Across Different LLMs (RQ3)

According to Section 5.4.3, o4-mini resolved most conflicts given P3 prompts. Thus, in this experiment, we sent P3 prompts to three additional LLMs, to observe how those models worked differently. As shown in Table 5.3 and Table 5.4, we compared models in two major aspects: effectiveness and cost.

Effectiveness Comparison: Table 5.3 shows that Sonnet-4 achieved the highest applicability (99%), followed by Flash (98%) and o4-mini (92%); Maverick showed the lowest applicability (76%). The three models with higher token limits (Sonnet-4, Flash, and Maverick) exceeded the token budget in only one conflict. Flash produced seemingly random, conflict-irrelevant outputs for 1 case, whereas Maverick produced meaningless outputs for 19 cases. The comparison implies that Maverick is insufficiently trained for conflict-handling tasks.

For conflict detection, Sonnet-4 obtained the highest F-score (99%), whereas Maverick again acquired the lowest (79%). For conflict resolution, Flash resolved the most conflicts (i.e., 59), while Maverick resolved the fewest (i.e., 21). Both Flash and Sonnet-4 resolved more conflicts than o4-mini.

Considering all effectiveness metrics, we noticed that Sonnet-4 and Flash generally outper-

formed o4-mini, which performed better than Maverick. Sonnet-4 is strongest at conflict detection, whereas Flash excels at conflict resolution. Notably, Flash was the only model to correctly resolve a particular conflict that none of the others well handled. As shown in Fig. 5.9, l renames the class `TypeSerializerConfig` to `SerializerConfig`, and simultaneously renames the file as well as constructor. Meanwhile, r adds a new class to use the original class and its constructor.

Fig. 5.9(3) shows an exemplar edit extracted from l, which adapts the type usage by systematically (1) replacing the class and constructor references, (2) renaming the local variable instantiated from that class (i.e., `typeSerializerConfig` → `serializerConfig`), and (3) updating the related literal (i.e., "`type-serializer`" → "`serializerConfig`"). This edit exactly matches developers' actual resolution edit for the added code. Flash successfully generated this ground-truth resolution, while the other models did not. More specifically, o4-mini and Sonnet-4 produced *partially* correct resolutions: they both updated the class usage, constructor invocation, and local variable name. However, neither of them produced a consistent update for the string literal. Maverick generated noisy output that was entirely unrelated to merge conflicts.

Cost Comparison: To assess the cost incurred by each model during conflict handling, we computed the total input tokens, total output tokens, and total API cost in US dollars. As shown in Table 5.4, Maverick was the most verbose, outputting the largest number of tokens (i.e., 117,305); o4-mini was the most succinct, producing the smallest number of tokens (i.e., 29,447). In terms of monetary cost, Sonnet-4 was the most expensive at $19.07 for the full dataset; Maverick was the least expensive at $0.76. There seems no strong correlation between the token volume and model cost.

Considering both effectiveness and cost, we found Flash to offer the best trade-off. It achieved the highest resolution accuracy at the second lowest cost. Although Sonnet-4 correctly

resolved the second most conflicts, its cost is 8–25 times of the cost by other models.

Finding 3: *Among the four models we used, Flash worked most effectively to resolve conflicts; Sonnet-4 was the most expensive, whereas Maverick was the lease expensive.*

5.5 Threats to Validity

Threats to External Validity Our observations may be limited to the build conflicts in our dataset, the prompt design we explored, and the LLMs we experimented with. To mitigate this threat and ensure the representativeness of our study, we tried our best to (1) include as much high-quality conflict data as possible into our dataset, (2) design and investigate six alternative prompt templates, (3) experiment with four state-of-the-art LLMs, and (4) use the default parameter settings of those LLMs, as recommended by the model creators. In the future, we plan to further mitigate this threat by experimenting with more data, more prompts, and more LLMs.

Threats to Internal Validity We evaluated each model's output by manually comparing it against developers' actual resolutions in the merged version m. Such a manual inspection can contain human errors, and may get limited by our domain knowledge. To mitigate these risks, three authors inspected all outputs simultaneously, and cross-checked their assessments. Whenever any discrepancy was observed, the authors discussed in depth until consensus was reached.

Threats to Construct Validity To ensure data quality, we required each merging scenario in our dataset to satisfy two conditions: (i) both branch versions compile successfully, and (ii) the automatically merged version A_m generated by git-merge contains no textual

conflict. In practice, some merging scenarios exhibit textual conflicts in A_m alongside build conflicts. These cases are excluded by condition (ii). As a result, our study does not capture any differences between build conflicts that co-occur with textual conflicts, and those that do not; nor does it capture potential differences in how such conflicts are resolved. In the future, we plan to include merging scenarios that satisfy (i) but violate (ii), to study whether our findings change under these more complex circumstances.

5.6 Summary

This paper presents the first systematic study of large language models for resolving build conflicts in Java programs. Unlike prior work that either relies on static analysis or applies few-shot learning to GPT-3, this study explores how well LLMs reason about build conflicts given prompts that have (1) minimized or full diffs of branch versions, and/or (2) domain knowledge encoded in the provided knowledge bases.

We observed interesting or even surprising phenomena. *Mini diff* consistently enables LLMs to detect conflicts accurately. In contrast, *full diff* is detrimental when used alone, yet becomes highly effective in conflict resolution when combined with mini diff in prompts. The 16 domain-specific *resolution rules* seldom help improve LLM performance; in fact, they can reduce effectiveness when models overuse them and derive incorrect resolutions. Project-specific *exemplar edits* provide limited benefit only when (1) the conflicts-to-resolve belong to the same merging scenario as the examples, (2) developers' resolutions closely resemble those examples, and (3) frequently used resolution rules learned by LLMs during pretraining are insufficient. Moreover, exemplar edits are less helpful than full diff when both are provided alongside mini diff, probably because full diff captures relevant edits more comprehensively, whereas example edits omit low-level details that are important for guiding

correct resolutions.

To investigate the trade-offs achieved by different LLMs, we also executed the best-performing template P3 using four alternative models. Our results show that Gemini 2.5 Flash achieves the best trade-off by acquiring the best result at the second lowest cost.

In future, we will explore automatic hybrid approaches that combine static analysis and automatic builds with LLMs. By using static analysis and build results to identify build conflicts and filter irrelevant diff information, we aim to improve LLM applicability, reduce usage cost, and preserve their strengths in conflict resolution.

Chapter 6

Conclusion

This research focuses on the detection and resolution of build conflicts in software merges. BUCOND, a static analysis tool that utilizes an interconnected graph structure and pattern matching, has been developed to effectively detect build conflicts. BUCOND demonstrated high precision and recall in revealing conflicts, particularly in scenarios where (1) textual and build conflicts coexist, or (2) compiler-based conflict detection is hindered by build errors triggered by initially revealed issues. BUCOND is the first static analysis-based tool to effectively detect Java build conflicts and has been evaluated using three separate datasets. Additionally, this work introduces a novel pattern set containing 57 types of conflict-triggering edit combinations. The design of this approach can be adapted to other object-oriented programming languages, enabling conflict detection in non-Java projects.

Building on the development of BUCOND, this research has focused on creating an approach to resolve build conflicts. To achieve this, **BuCoR** has been developed, utilizing a novel hybrid program transformation approach for suggesting build conflict resolutions. **BuCoR** has been evaluated on a real-world dataset and has demonstrated the ability to (i) generate syntactically correct resolutions for build conflicts, and (ii) suggest resolutions that align with those made by developers. This highlights **BuCoR**'s potential to assist developers in the conflict resolution process by providing accurate resolution edits. This work has also provided a study on the general conflict resolution patterns utilized by developers. This can

aid further investigation on this topic. Additionally, this work includes a study of common conflict resolution patterns used by developers, which can serve as a foundation for further research in this area.

The final phase of this research investigates the capability of LLMs for build conflict resolution. As LLMs become more common in software engineering, it is important to understand where they can meaningfully improve developer productivity. In this work, we examined how different prompt designs shape the reasoning ability of a default LLM and then extended the analysis to several state-of-the-art models. Our results show that prompt structure plays a major role in helping LLMs infer and repair semantic inconsistencies in merged code, and that higher model cost does not always translate to better solutions. We have demonstrated that when properly guided, LLMs can resolve build conflicts with high effectiveness, underscoring the value of well-designed prompts and context. Overall, these findings suggest that LLM-based approaches can complement static analysis techniques and offer a promising direction for future work on automated merge conflict resolution.

Future work will extend this investigation in several directions. We plan to integrate insights from static analysis as additional structured context to form a hybrid static–LLM framework, as well as extend the framework to incorporate richer project-level semantics. Ultimately, we envision adaptive merge systems that unify multiple sources of semantic and contextual information to support more general, robust, and scalable conflict resolution.

Bibliography

[1] Git-merge documentation.

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

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

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

[5] jFSTMerge. https://github.com/guilhermejccavalcanti/jFSTMerge, 2021.

[6] Maven. https://maven.apache.org, 2021.

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

[8] Activiti. https://github.com/Activiti/Activiti, 2022.

[9] druid. https://github.com/alibaba/druid, 2022.

[10] elasticsearch. https://github.com/elastic/elasticsearch, 2022.

[11] fastjson. https://github.com/alibaba/fastjson, 2022.

[12] Feature Branching: A Guide to the Do's and Don'ts. https://launchdarkly.com/blog/dos-and-donts-of-feature-branching/, 2022.

[13] hazelcast. https://github.com/hazelcast/hazelcast/commit/725d5235cbd6835c308b2e819201782301813842, 2022.

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

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

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

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

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

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

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

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

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

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

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

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

[26] WALA. https://github.com/wala/WALA, 2024.

[27] Gemini 2.5 Flash - Google DeepMind. https://deepmind.google/models/gemini/flash/, 2025.

[28] Introducing Claude 4. https://www.anthropic.com/news/claude-4, 2025.

[29] JavaParser. https://javaparser.org, 2025.

[30] JGraphT. https://jgrapht.org, 2025.

[31] Merge branch '3.0' of `github.com:hazelcast/hazelcast` into 3.0. https://github.com/hazelcast/hazelcast/commit/dacc16c1860d646f4b7d6d921bd4438b35d899ae, 2025.

[32] Merge branch 'mrniko/master' into test-timeout. https://github.com/redisson/redisson/commit/9baf319ecb41dfc42d273d467d8f55ed2ba6daa7, 2025.

[33] Merge commit in `swagger-core`: feature/jaxrs2_reader_oas_v3.0.0. https://github.com/swagger-api/swagger-core/commit/9c38329c20ae27c6680d5833c68b07b85f512dd4, 2025.

[34] Merge pull request #106 from ptma/master. https://github.com/alibaba/fastjson/commit/7a56c582f6de20a7a775d48f1aa0d874f2c0206c, 2025.

[35] Merge pull request #917 from argvk/spop_with_count. https://github.com/redis/jedis/commit/30986c51de6d914a1f10f620613674af017c65ea, 2025.

[36] O4-mini. https://platform.openai.com/docs/models/o4-mini, 2025.

[37] The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/, 2025.

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

[39] Git Merge. https://git-scm.com/docs/git-merge, Last visited 07/26/2019.

[40] Iftekhar Ahmed, Caius Brindescu, Umme Ayda Mannan, Carlos Jensen, and Anita Sarma. 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)*, pages 58–67. IEEE, 2017.

[41] Juan Altmayer Pizzorno and Emery D. Berger. Coverup: Effective high coverage test generation for python. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025.

[42] Sven Apel, Olaf Lessenich, and Christian Lengauer. Structured merge with auto-tuning: Balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 120–129, New York, NY, USA, 2012. ACM.

[43] Sven Apel, Jorg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kastner. 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*, ESEC/FSE '11, pages 190–200, New York, NY, USA, 2011. ACM.

[44] Sven Apel, Jörg Liebig, Christian Lengauer, Christian Kästner, and William R Cook. Semistructured merge in revision control systems. In *VaMoS*, pages 13–19, 2010.

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

[46] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 257–268, New York, NY, USA, 2014. Association for Computing Machinery.

[47] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering*, 25(1):562–590, 2020.

[48] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering*, 25(1):562–590, 2020.

[49] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[50] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, 39(10):1358–1375, Oct 2013.

[51] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 168–178, New York, NY, USA, 2011. ACM.

[52] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. Evaluating and improving semistructured merge. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[53] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. The impact of structure on software merging: Semistructured versus structured merge. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, pages 1002–1013. IEEE Press, 2019.

[54] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineer-*

ing, FSE 2024, page 572–576, New York, NY, USA, 2024. Association for Computing Machinery.

[55] Léuson Da Silva, Paulo Borba, and Arthur Pires. Build conflicts in the wild. *Journal of Software: Evolution and Process*, 34(4):e2441, 2022.

[56] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu K Lahiri. Deepmerge: Learning to merge programs. *arXiv preprint arXiv:2105.07569*, 2021.

[57] Joachim Eibl. The KDiff3 Handbook. https://kdiff3.sourceforge.net/doc/index.html.

[58] Paulo Elias, Heleno de S. Campos, Eduardo Ogasawara, and Leonardo Gresta Paulino Murta. Towards accurate recommendations of merge conflicts resolution strategies. *Information and Software Technology*, 164:107332, 2023.

[59] H Christian Estler, Martin Nordio, Carlo A Furia, and Bertrand Meyer. Awareness and merge conflicts in distributed software development. In *2014 IEEE 9th International Conference on Global Software Engineering*, pages 26–35. IEEE, 2014.

[60] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin 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*, pages 313–324, 2014.

[61] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023.

[62] Gleiph Ghiotto, Leonardo Murta, Marcio Barros, and Andre Van Der Hoek. 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):892–915, 2018.

[63] Mário Luís Guimarães and António Rito Silva. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 342–352, Piscataway, NJ, USA, 2012. IEEE Press.

[64] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*, 2023.

[65] Zijian Jiang, Hao Zhong, and Na Meng. Investigating and recommending co-changed entities for javascript programs. *Journal of Systems and Software*, 180:111027, 2021.

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

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

[68] Hoai Le Nguyen and Claudia-Lavinia Ignat. An analysis of merge conflicts and resolutions in git-based open source projects. *Computer Supported Cooperative Work (CSCW)*, 27(3):741–765, 2018.

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

[70] Olaf Leßenich, Janet Siegmund, Sven Apel, Christian K'astner, and Claus Hunsen. Indicators for merge conflicts in the wild: Survey and empirical study. *Automated Software Engg.*, 25(2):279–313, June 2018.

[71] Kyla H. Levin, Nicolas van Kempen, Emery D. Berger, and Stephen N. Freund. Chatdbg: Augmenting debugging with large language models. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025.

[72] Yihe Li, Ruijie Meng, and Gregory J. Duck. Large language model powered symbolic execution. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025.

[73] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. Prompt fuzzing for fuzz driver generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 3793–3807, New York, NY, USA, 2024. Association for Computing Machinery.

[74] Chandra Maddila, Nachiappan Nagappan, Christian Bird, Georgios Gousios, and Arie van Deursen. Cone: A concurrent edit detection tool for large scalesoftware development. *arXiv preprint arXiv:2101.06542*, 2021.

[75] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. 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)*, pages 151–162. IEEE, 2019.

[76] Yacine Majdoub and Eya Ben Charrada. Debugging with open-source large language models: An evaluation. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '24, page 510–516, New York, NY, USA, 2024. Association for Computing Machinery.

[77] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[78] Dimitrios Michail, Joris Kinable, Barak Naveh, and John V Sichi. Jgrapht—a java library for graph data structures and algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 46(2):1–29, 2020.

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

[80] Thu-Trang Nguyen, Thanh Trong Vu, Hieu Dinh Vo, and Son Nguyen. An empirical study on capability of large language models in understanding code semantics. *arXiv preprint arXiv:2407.03611*, 2024.

[81] Anh Nguyen-Duc, Beatriz Cabrero-Daniel, Adam Przybylek, Chetan Arora, Dron Khanna, Tomas Herda, Usman Rafiq, Jorge Melegati, Eduardo Guerra, Kai-Kristian Kemell, et al. Generative artificial intelligence for software engineering–a research agenda. *arXiv preprint arXiv:2310.18648*, 2023.

[82] OpenAI. Chatgpt: A large language model. https://chat.openai.com/, 2022. Accessed: 2024-08-12.

[83] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. Predicting merge conflicts in collaborative software development. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[84] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu Lahiri, and Mike 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)*, pages 785–796. IEEE, 2021.

[85] Duarte Guedes Sardão. Preliminary study on the effectiveness of chatgpt at identifying, explaining, and revealing semantic conflicts in merge commits. 2024.

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

[87] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. Intellimerge: A refactoring-aware software merging technique. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, 2019.

[88] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. Intellimerge: A refactoring-aware software merging technique. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[89] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. Intellimerge: A refactoring-aware software merging technique. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[90] Bowen Shen, Muhammad Ali Gulzar, Fei He, and Na Meng. A characterization study of merge conflicts in java projects. *ACM Trans. Softw. Eng. Methodol.*, jun 2022. Just Accepted.

[91] Bowen Shen, Muhammad Ali Gulzar, Fei He, and Na Meng. A characterization study of merge conflicts in java projects. *ACM Trans. Softw. Eng. Methodol.*, jun 2022. Just Accepted.

[92] Bowen Shen and Na Meng. Conflictbench: A benchmark to evaluate software merge tools. *Journal of Systems and Software*, 214:112084, 2024.

[93] Bowen Shen, Cihan Xiao, Na Meng, and Fei He. Automatic detection and resolution of software merge conflicts: Are we there yet? *arXiv preprint arXiv:2102.11307*, 2021.

[94] Chaochao Shen, Wenhua Yang, Minxue Pan, and Yu Zhou. Git merge conflict resolution leveraging strategy classification and llm. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pages 228–239. IEEE, 2023.

[95] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 23–30. IEEE, 2023.

[96] Marcelo Sousa, Isil Dillig, and Shuvendu Lahiri. Verified three-way program merge. In *Object-Oriented Programming, Systems, Languages & Applications Conference (OOPSLA 2018)*. ACM, November 2018.

[97] Chungha Sung, Shuvendu K. Lahiri, Mike Kaufman, Pallavi Choudhury, and Chao Wang. Towards understanding and fixing upstream merge induced conflicts in divergent forks: An industrial case study. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 172–181, 2020.

[98] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. Is chatgpt the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*, 2023.

[99] Sheikh Shadab Towqir, Bowen Shen, Muhammad Ali Gulzar, and Na Meng. Detecting

build conflicts in software merge for java programs via static analysis. ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[100] Chuck Walrad and Darrel Strom. The importance of branching models in scm. *Computer*, 35(9):31–38, 2002.

[101] Thorsten Wuensche, Artur Andrzejak, and Sascha Schwedes. Detecting higher-order merge conflicts in large software projects. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 353–363, 2020.

[102] Thorsten Wuensche, Artur Andrzejak, and Sascha Schwedes. Detecting higher-order merge conflicts in large software projects. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 353–363. IEEE, 2020.

[103] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025.

[104] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[105] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494, 2023.

[106] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

[107] Ryohei Yuzuki, Hideaki Hata, and Kenichi Matsumoto. How we resolve conflict: an

empirical study of method-level conflict resolution. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, pages 21–24. IEEE, 2015.

[108] Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K Lahiri. Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 77–88, 2022.

[109] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 913–923, 2015.

[110] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Trans. Softw. Eng. Methodol.*, 34(5), May 2025.

[111] Fengmin Zhu and Fei He. Conflict resolution for structured merge via version space algebra. *Proc. ACM Program. Lang.*, 2(OOPSLA):166:1–166:25, October 2018.