UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

João Pedro Henrique Santos Duarte

The Impact of Language Independence on Structured Merge Accuracy and Efficiency

Recife

2025

João Pedro Henrique Santos Duarte

The Impact of Language Independence on Structured Merge Accuracy and Efficiency

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software

Orientador (a): Prof. Dr. Paulo Henrique Monteiro Borba

Coorientador (a): Prof. Dr. Guilherme José de Carvalho Cavalcanti

Recife

2025

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Duarte, João Pedro Henrique Santos.
   The impact of language independence on structured merge accuracy and efficiency / João Pedro Henrique Santos Duarte. - Recife, 2025.
   71f.: il.

   Dissertação (Mestrado)- Universidade Federal de Pernambuco, Centro de Informática, Programa de Pós-Graduação em Ciência da Computação, 2025.
   Orientação:Paulo Henrique Monteiro Borba.

   1. Integração de código; 2. Ferramentas de merge; 3. Merge estruturado; 4. Linguagens de programação. I. Borba, Paulo Henrique Monteiro. II. Título.

UFPE-Biblioteca Central

UNIVERSIDADE FEDERAL DE PERNAMBUCO

Ata da defesa/apresentação do Trabalho de Conclusão de Curso de Mestrado do Programa de Pós-graduação em Ciências da Computação - CIN da Universidade Federal de Pernambuco, no dia 29 de julho de 2025.

ATA Nº 2220

Ao vigésimo nono dia do mês de julho do ano de dois mil e vinte e cinco, às dez horas, no Centro de Informática da Universidade Federal de Pernambuco, teve início a duas milésima ducentésima vigésima defesa de dissertação do Mestrado em Ciência da Computação, intitulada The Impact of Language Independence on Structured Merge Accuracy and Efficiency, na área de concentração de Engenharia de Software e Linguagem de Programação, do candidato João Pedro Henrique Santos Duarte o qual já havia preenchido anteriormente as demais condições exigidas para a obtenção do grau de mestre. A Banca Examinadora, composta pelos professores André Luis de Medeiros Santos, pertencente ao Centro de Informática desta Universidade, Melina Mongiovi Brito Lira, pertencente a Unidade Acadêmica de Sistemas e Computação da Universidade Federal de Campina Grande, e Paulo Henrique Monteiro Borba, pertencente ao Centro de Informática desta Universidade, sendo o primeiro presidente da banca examinadora e o último orientador do trabalho de dissertação, decidiu: Aprovar o trabalho. E para constar lavrei a presente ata que vai por mim assinada e pela Banca Examinadora.

Dra. MELINA MONGIOVI BRITO LIRA, UFCG

Examinadora Externa à Instituição

Dr. PAULO HENRIQUE MONTEIRO BORBA, UFPE

Examinador Interno

Dr. ANDRE LUIS DE MEDEIROS SANTOS, UFPE

Presidente

JOAO PEDRO HENRIQUE SANTOS DUARTE

Mestrando(a)

Dedico a todos que, de alguma forma, contribuíram para a realização deste trabalho.

AGRADECIMENTOS

Agradeço primeiramente a minha família: minha companheira Laura e meus pais Henrique e Edlamar. As discussões e o carinho que recebi foram essenciais para que eu pudesse concluir este trabalho com sucesso. Agradeço mais ainda pela compreensão nos momentos em que precisei me ausentar para me dedicar a este projeto.

Aos meus amigos e colegas que sempre estiveram disponíveis durante toda essa jornada. Seria impossível citar todos vocês, mas vossas contribuições foram fundamentais para o meu crescimento em todas as esferas.

Ao meu orientador, Prof. Paulo Borba, pela orientação, paciência e apoio durante todo o processo deste trabalho. Para além da academia, as conversas que tivemos foram muito valiosas e me ajudaram bastante a seguir firme nesse caminho árduo da pesquisa.

Ao meu co-orientador, Prof. Guilherme Cavalcanti pelo apoio e pelas discussões que contribuíram significativamente para o desenvolvimento deste trabalho.

À banca pela disponibilidade em avaliar e contribuir com valiosas ponderações que certamente enriquecerão este trabalho.

Aos membros do SPG *(Software Productivity Group)*, pelos momentos e discussões que sempre foram muito enriquecedoras e resultaram em valiosas contribuições a este trabalho.

Por fim, agradeço ao CIn e à UFPE pelas experiências que tive aqui e todas as oportunidades que me foram proporcionadas e que moldaram minha trajetória acadêmica e profissional.

RESUMO

Ferramentas de *merge* não estruturadas são amplamente utilizadas na prática. Ferramentas de *merge* estruturadas baseadas em ASTs apresentam uma precisão de *merge* significativamente melhor, mas são específicas para cada linguagem e custosas, o que faz com que não estejam disponíveis para muitas linguagens de programação. Essa restrição limita a adoção de ferramentas estruturadas na indústria, já que muitas equipes trabalham com múltiplas linguagens e não podem arcar com a manutenção de uma ferramenta de *merge* separada para cada uma delas. Para melhorar a precisão do *merge* em uma ampla variedade de linguagens, propomos LastMerge, uma ferramenta de *merge* estruturada genérica que pode ser configurada por meio de uma interface simples, reduzindo significativamente o esforço necessário para dar suporte ao *merge* estruturado. Para entender o impacto que uma ferramenta de *merge* estruturada genérica pode ter na precisão e no desempenho do merge, conduzimos um experimento com quatro ferramentas de *merge* estruturado: duas específicas para Java, *jDime* e *Spork*, e suas contrapartes genéricas, respectivamente LastMerge e *Mergiraf*. Utilizando cada ferramenta, reexecutamos cenários de *merge* de um conjunto de projetos significativo e coletamos dados sobre tempo de execução, divergências comportamentais e precisão do merge. Nossos resultados mostram que não há evidências de que o *merge* estruturado genérico impacte significativamente a precisão do merge. Embora observemos uma taxa de diferença de aproximadamente 10% entre as ferramentas específicas para Java e suas contrapartes genéricas, a maioria das diferenças decorre de detalhes de implementação e poderia ser evitada. LastMerge reporta 15% menos falsos positivos (conflitos espúrios) que o *jDime*, enquanto o *Mergiraf* deixa de identificar 42% menos falsos negativos (conflitos reais ignorados) que o *Spork*. Ambas as ferramentas genéricas apresentam desempenho de tempo de execução comparável às implementações específicas por linguagem mais avançadas. Também exploramos o esforço necessário para configurar LastMerge para uso com linguagens de programação, configurando-o para uso com Java e C#. Verificamos que o esforço é significativamente menor do que o necessário para manter uma ferramenta de *merge* específica por linguagem, exigindo apenas um conhecimento mínimo da estrutura da linguagem de programação. Além disso, essa configuração pode ser melhorada de forma incremental ao longo do tempo, conforme a ferramenta é usada na prática, para aprimorar a precisão do *merge* nos diversos cenários encontrados. Esses resultados sugerem que ferramentas de *merge* estruturadas genéricas podem

substituir efetivamente as ferramentas específicas por linguagem, abrindo caminho para uma adoção mais ampla do *merge* estruturado na indústria.

Palavras-chave: Integração de código. Ferramentas de *Merge*. *Merge* estruturado. Linguagens de Programação

ABSTRACT

Unstructured line-based merge tools are widely used in practice. Structured AST-based merge tools show significantly improved merge accuracy, but are language specific and costly, consequently not being available for many programming languages. Such restriction limits the adoption of structured merge tools in industry, as many teams work with multiple programming languages and cannot afford to maintain a separate merge tool for each language. To improve merge accuracy for a wide range of languages, we propose LastMerge, a *generic* structured merge tool that can be configured through a thin interface that significantly reduces the effort of supporting structured merge. To understand the impact that *generic* structured merge might have on merge accuracy and performance, we run an experiment with four structured merge tools: two Java specific ones, *jDime* and *Spork*, and their *generic* counterparts, respectively LastMerge and *Mergiraf*. Using each tool, we replay merge scenarios from a significant dataset, and collect data on runtime, behavioral divergences, and merge accuracy. Our results show no evidence that *generic* structured merge significantly impacts merge accuracy. Although we observe a difference rate of approximately 10% between the Java specific tools and their *generic* counterparts, most of the differences stem from implementation details and could be avoided. We find that LastMerge reports 15% fewer false positives than *jDime* while *Mergiraf* misses 42% fewer false negatives than *Spork*. Both *generic* tools exhibit comparable runtime performance to the state of the art language specific implementations. We also explore the effort of configuring LastMerge for usage with programming languages, by configuring it for usage with Java and C#. We find that the effort is significantly lower than maintaining a language-specific merge tool, requiring only minimal knowledge of the programming language structure. Furthermore, such configuration can be incrementally improved over time, as the tool is used in practice, in order to improve merge accuracy for the variety of scenarios encountered in practice. These results suggest that *generic* structured merge tools can effectively replace language-specific ones, paving the way for broader adoption of structured merge in industry.

Keywords: Code integration. Merge tools. Structured merge. Programming languages.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUCTION

In most projects, developers collaborate by working on separate branches or repositories (like local and remote ones), and later merge their changes into the main codebase. Version Control Systems (VCSs) typically rely on line-based, *unstructured*, merge tools such as *diff3* (MENS, 2002). These tools compare file revisions based solely on the textual content of their lines, without considering the syntactic or semantic structure of the code (KHANNA; KUNAL; PIERCE, 2007). Such unstructured merge techniques are fast, language-agnostic, and widely used in practice. However, they often produce spurious conflicts that waste developer effort to fix semantically harmless issues that could be otherwise automatically resolved. At the same time, unstructured merge may overlook actual conflicts, which can silently propagate into final artifacts and cause regressions in production.

To improve the accuracy of merging software revisions, researchers have proposed tools that leverage the syntactic structure of the source code (HUNT; TICHY, 2002; APEL et al., 2011; ZHU; HE; YU, 2019; WESTFECHTEL, 1991; CLEMENTINO; BORBA; CAVALCANTI, 2021; BUFFENBARGER, 1995; LARSEN et al., 2023; APEL; LEßENICH; LENGAUER, 2012). Unlike unstructured tools, *structured* merge tools parse the source code into an Abstract Syntax Tree (AST) and apply tree matching and combination algorithms to generate the merged artifact. Prior studies have shown that structured merge significantly improves merge accuracy compared to unstructured techniques (SEIBT et al., 2022; SCHESCH et al., 2024). Despite these advances, structured tools remain largely absent from current practice especially for two reasons. First, as they strongly rely on the syntax and semantics of specific programming languages, structured tools proposed so far are language specific. Second, for being costly, not many languages are supported by structured tools. State of the art tools, for example, are typically designed for usage with only Java (APEL; LEßENICH; LENGAUER, 2012; LARSEN et al., 2023). This means that substantial implementation and maintenance effort would needed for supporting a new language. On the other hand, software projects in industry often involve multiple programming languages, which are typically used for different purposes. Being restricted to a single language, structured merge tools cannot be used in such polyglot projects, which limits their applicability in practice.

To reduce these barriers and improve merge accuracy for a wide range of programming languages, we introduce LastMerge, a *generic* structured merge tool that can be easily

configurable for each language. It relies on a core merge engine that operates over generic trees produced by an extensible and fast parser framework (BRUNSFELD, 2025) that has been instantiated for more than 350 languages, and is in production at GitHub. By feeding the engine with a high level description of language specific aspects (such as node labelling and restrictions to permutation of node children) that are known to be relevant for structured merge, developers can easily adapt LastMerge for new languages, or refine support for existing ones. This thin configuration interface significantly reduces the effort of having structured merge for a wide range of languages.

Apart from its implementation, this work also investigates the potential of LastMerge for usage in production software projects. To do this, we divide our investigation into two main parts. The first part aims to understand how *generic* structured merge tools compares to the existing language specific state of the art ones. We aim to understand wether *generic* structured merge can act as a drop-in replacement for language-specific structured merge tools. In the second part, we evaluate the generalization capabilities of LastMerge. We do this by investigating the effort involved in instantiating LastMerge for usage with a programming language other than Java.

To understand the impact that a *generic* structured merge technique might have on merge accuracy and computational performance, we run a comparative experiment with four structured merge tools, paired as follows: *jDime* (APEL; LEßENICH; LENGAUER, 2012), a well-known Java specific tool, and LastMerge, which adopts a similar algorithm but on top of a language independent AST and *generic* setting; and *Spork* (LARSEN et al., 2023), a more recent Java specific tool, and *Mergiraf*,[1] which adopts a similar algorithm but on top of the same language independent setting as LastMerge. This pairing reflects the counterpart influence of existing state of the art, language specific, tools on the design and implementation of the generic ones. We replay merge scenarios using each tool (the generic ones instantiated with Java syntactic and semantic details) in a significant dataset (SCHESCH et al., 2024), and collect data on runtime, behavioral divergences, and merge accuracy. Specifically, we compute the number of spurious conflicts (false positives) and actual missed conflicts (false negatives). We address the following research questions: How *generic* structured merge impacts merge accuracy? How *generic* structured merge impacts merge runtime performance?

Our results show no evidence that *generic* structured merge significantly impacts merge accuracy. Although we observe a difference rate of approximately 10% between the Java spe-

[1] <https://mergiraf.org/>

cific tools (*jDime* and *Spork*) and their *generic* counterparts (LastMerge and *Mergiraf*) instantiated for Java, most of the differences stem from implementation details and configuration choices, not from design decisions implied by the generality requirement. We also find that LastMerge reports 15% fewer false positives than *jDime*, while *Mergiraf* misses 42% fewer false negatives than *Spork*. Furthermore, both *generic* tools exhibit comparable runtime performance to the state of the art language-specific implementations.

Finally, with the goal of understanding the effort envolved in configuring LastMerge for new languages, we also conduct a case study where the author instantiates a vanila configuration for usage with C#. We argue that the effort is relatively simple, such that a user with fairly basic knowledge of the language and some high level knowledge of the technologies used could derive a similar minimal configuration within at most a few days. This is significantly less than the expected effort for implementing a new language-specific tool, which would involve adapting or even reimplementing several algorithms — an effort that may take weeks or even months. These results suggest that LastMerge can effectively replace language-specific tools, achieving similar levels of accuracy and efficiency while easing the onboarding of new languages, thus paving the way for a broader adoption of structured merge in industry.

The rest of this document is organized as follows. In Chapter 2, we present the essential concepts used in this work and motivates the need for language independent structured merge tools. Chapter 3 discusses the implementation of language independent structured merge tools. We present the design and implementation decisions of LastMerge, our proposed tool, and briefly discuss how it compares to the other tools used in our experiment. In Chapter 4 we present our empirical study that evaluates the impact of *generic* structured merge on merge accuracy and performance. We present our research questions, sampling process, and methodology. We also present the results of our experiment and discuss how they help us answer our research questions. Chapter 5 presents the configuration interface for LastMerge. This configuration allows users to easily adapt the tool for new languages or refine support for existing ones. We also present a case study where we instantiate LastMerge for C#. Finally, Chapter 6 concludes this document. We summarize our contributions and discuss both related and future work.

2 BACKGROUND

In this chapter, we explain the main concepts used on this work. First, Section 2.1 discusses the fundamentals of Version Control Systems (VCSs), which are the main supporting tools of modern collaborative development. Section 2.2 introduces and contrast the strategies of unstructured and structured merge through an illustrative example, as well as discuss the costs of supporting structured merge for different languages. Lastly, Section 2.3 presents the two state of the art structured merge tools available for Java used in our study: *jDime* and *Spork*.

2.1 VERSION CONTROL SYSTEMS (VCS)

With the ever rising complexity of software projects, development has become an increasingly collaborative effort. Within this context, concepts such as *software configuration management (SCM)* and *Version Control Systems (VCS)* naturally arose to support collaborative software development. *SCM* is often related in a broader sense to the management of software artifacts, especially in environments where multiple versions of it are maintained. (CONRADI; WESTFECHTEL, 1998) *VCS*, on the other hand, are a particular approach for handling changes and updates to software artifacts, while ensuring the share and consistency of these artifacts among multiple developers and environments. Such *VCS* systems are often classified into two main categories: *centralized* and *distributed* version control systems. Figure 1 illustrates a high level view of the differences between each architecture.

Centralized Version Control Systems (CVCS) are characterized by a single central repository that stores all versions of the project files. On this model, developers updates their particular repositories by checking out files from the central repository before doing contributions. Once they have made changes, they check the files back in to the central repository. This process ensures that all developers are working with the same version of the project files, aiming to keep the current development stable and accessible for all involved. Many services implement this approach, including Subversion (SUBVERSION, 2025) and CVS (CVS, 2025).

Decentralized Version Control Systems (DVCS) instead, foments developers to have their own copies of the main repository. Each particular repository can be used as the source of information for project history. These different repositories can be synchronized with each other, allowing developers to share their changes and updates. This approach allows for greater

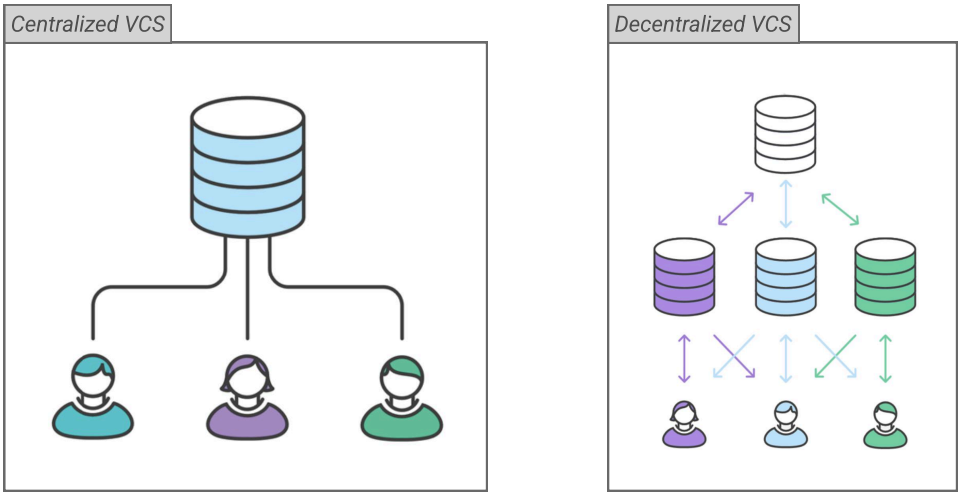

Figura 1 – The image highlights the differences between centralized and distributed version control systems. In a *centralized* system, developers interact with a single central repository, while in a *distributed* system, each developer has their own repository that can be synchronized with others. The image is based on (CAVALCANTI, 2019)

flexibility and independence, as developers can work on their own copies of the project without needing to be connected to a central server. Popular examples of *DVCS* include Git (GIT, 2025) and Mercurial (MERCURIAL, 2025).

In comparison with the vertical contribution style of *CVCS*, *DVCS* empowers developers to work in a more horizontal manner, where they can contribute to the project in parallel without being constrained by a central authority. (RIGBY et al., 2009) Such difference has led to a shift in the way software development is approached, with *DVCS* becoming the preferred choice for many open source projects and collaborative software development environments. GitHub, a popular platform that uses Git as its underlying *DVCS*, facilitates collaboration and version control among developers that powered more than 1 billion contributions in 2024 (GITHUB, 2024).

Despites the differences between *CVCS* and *DVCS*, both systems share the same core principle of allowing software developers to create parallel versions of software. This allows developers to work on different features or bug fixes without interfering with each other's work. The core challenge of these systems is to ensure that changes made by different developers can be merged together in a consistent and reliable manner to form one final version. This is where the concept of *merge* comes into play, as it allows developers to combine their changes and

updates into a single version of the project. Several techniques and tools have been developed to support this process, ranging from simple unstructured merge tools to more sophisticated structured merge tools that take into account the syntax and semantics of the code being merged.

2.2 UNSTRUCTURED AND STRUCTURED MERGE

To illustrate the differences between unstructured and structured merge, and the costs of supporting the latter for a number of languages, consider the merge scenario illustrated in Figure 2. It shows the initial declaration of the `debit` method in the `Account` class. Starting from this base version, two developers, *Left* and *Right*, independently modify the method. *Left* makes the method `public`, while *Right* makes the method `static`. As these changes differ, they must be merged to produce the final version of the `Account` class.

In this scenario, an unstructured merge tool such as *diff3* (KHANNA; KUNAL; PIERCE, 2007) performs a line-based comparison between the revisions, using the common base version as a reference. Since both developers modify the same line, the tool is unable to integrate the changes and reports a conflict, as illustrated in Figure 3. Resolving this conflict requires manual intervention to combine the modifications to preserve developers intentions.

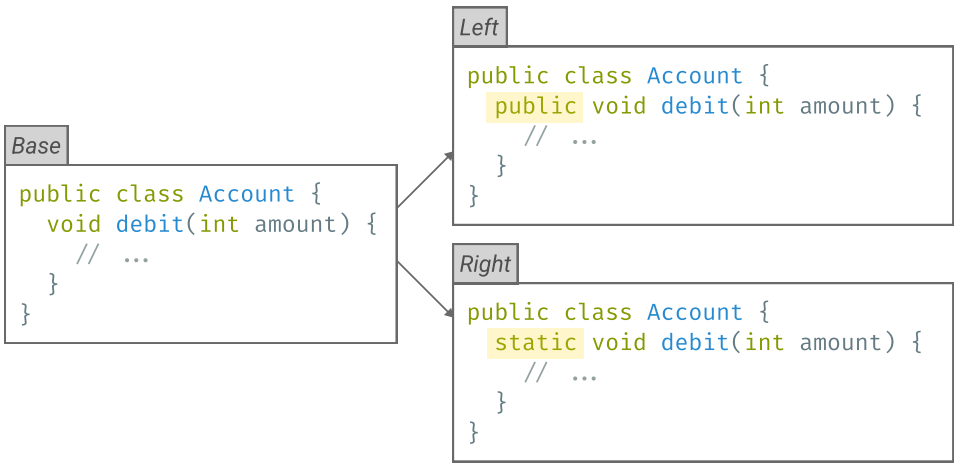

Figura 2 – An example of a merge scenario. *Base* shows the initial revision, or the shared common ancestor. Both *Left* and *Right* are revisions that independently introduce changes to *Base*. Changes are highlighted in yellow.

```
public class Account {
<<<<<<<
  public void debit(int amount) {
=======
  static void debit(int amount) {
>>>>>>>
    // ...
  }
}
```

Figura 3 – The output of unstructured merge. Since it only relies on the textual content of the lines, a conflict between the changes introduced by *Left* and *Right* is reported, as presented in red.

As shown in Figure 4, structured merge correctly merges *Left* and *Right* changes avoiding developer effort. Structured tools (HUNT; TICHY, 2002; APEL et al., 2011; ZHU; HE; YU, 2019; WESTFECHTEL, 1991; CLEMENTINO; BORBA; CAVALCANTI, 2021; BUFFENBARGER, 1995; LARSEN et al., 2023; APEL; LEßENICH; LENGAUER, 2012) are language-specific and leverage language syntax and semantics. In the Java case, they explore the fact that modifiers such as `public` and `static` can be applied together and in any order. Furthermore, they can also enforce language rules to prevent the generation of semantically incorrect code. For example, the tool can ensure that the resulting method after integration is not declared both `public` and `private`, which would violate Java's language specifications. Instead of comparing lines of text, these tools firstly construct tree representations of the source code— typically Abstract Syntax Trees (ASTs)— for each revision to be merged. During the *matching* phase, the tool correlates common and modified nodes across revisions. In the subsequent *amalgamation* phase, it merges the nodes based on the collected matching information. Conflicts are reported only when different changes affect corresponding tree nodes. Structured merge also avoids spurious conflicts when, for instance, developers independently add declarations (field, method, etc.) to the same area of the text, or change syntactically separate parts of an expression or statement, even if they appear in the same or consecutive lines of code.

```
public class Account {
  public static void debit(int amount) {
    // ...
  }
}
```

Figura 4 – The output of structured merge. Since the changes made by *Left* and *Right* occur in different tree nodes, no conflict is reported.

Although structured merge outperforms unstructured merge by reporting fewer spurious conflicts (SEIBT et al., 2022; SCHESCH et al., 2024), and even detecting conflicts that are missed by unstructured tools, creating such a tool demands significant effort for each language that needs to be supported. Besides creating or adapting[1] language-specific parsers and ASTs, one has to implement the whole merging (matching, amalgamation, etc.) engines for each AST, which is often expensive. Maintaining such engines is also costly, as they need to be fixed or updated for each supported language. This explains why structured merge tools are available for only a few languages, and have not been widely adopted in industry, where tools that support multiple languages are often needed for most nontrivial projects.

2.3 JAVA SPECIFIC STRUCTURED TOOLS

Given the difficulties of implementing structured merge tools for general-purpose programming languages, most of the existing work focused on specific languages, such as Java. In this context, two contenders arise as the state of the art structured tools for structured merging: *jDime* (APEL; LEßENICH; LENGAUER, 2012) and *Spork* (LARSEN et al., 2023). Both are publicly available on GitHub, and we used them in our empirical study, as detailed in Chapter 4. This section provides a brief overview of them.

jDime was proposed by Apel, Leßenich and Lengauer (2012) and still arises as one of the most mature and benchmarked implementations of structured merge for Java. It implements a three-way structured merge algorithm that operates on the Abstract Syntax Tree (AST) representation of Java source code. By using a parser implemented on top of JastAddJ, it first parses the three revisions of the merge scenario into their corresponding ASTs. Then, it computes the differences between the base AST and each of the modified ASTs using a combination of tree matching algorithms. Finally, it merges the changes from both modified ASTs into a single AST, resolving conflicts based on the structure of the code. We have reimplemented these algorithms on our tool, LastMerge, and we dive deep into the details of them in Chapter 3.

More recently, *Spork* was proposed by Larsen et al. (2023) as another structured merge tool for Java. *Spork* focuses on maintaining code readability by implementing a high-fidelity pretty-printing mechanism that aims to preserve the original formatting of the code. Other

[1] Tools often reuse existing parser infrastructure: *Spork* relies on Spoon (LARSEN et al., 2023), and *jDime* on JastAddJ (APEL; LEßENICH; LENGAUER, 2012).

existing tools, such as *jDime*, often struggle with this aspect, leading to merged code that is syntactically correct but that might not respect the original style and formatting of the code.

Spork employs the 3DM-MERGE algorithm (LARSEN et al., 2023) to merge revisions. The algorithm operates by firstly encoding input trees as a set of *Parent-Child-Successor* (PCS) triples. Each triple (p, c, s), where p is a node, indicates that both c and s are children of p, with s immediately succeeding c in p's children list (LARSEN et al., 2023). A set of PCS triples is referred to as a *changeset*. Changesets from each revision are combined into a single changeset to construct the final merged tree. However, inconsistencies may arise during this process. For example, if the base contains a PCS triple (x, y, z) and a revision adds a new PCS triple (x', y, z), the changeset becomes inconsistent because nodes y and z now have different parents. *Mergiraf* heuristically resolves some of these inconsistencies by removing triples from the base revision until the changeset becomes consistent.

However, even after applying these heuristics, some inconsistencies may persist. In most cases, these arise because the changes introduced by the revisions truly conflict, so the tool reports a conflict. In other cases, the algorithm resolves inconsistencies by applying heuristics such as reordering a node's children, as illustrated by the reordering of method modifiers in Figure 4.

Both *jDime* and *Spork* have been empirically evaluated and compared against unstructured merge tools, showing that they can reduce the number of spurious conflicts in real-world merge scenarios (APEL; LEßENICH; LENGAUER, 2012; LARSEN et al., 2023). Previous work also compared them against each other, showing that they have different strengths and weaknesses (LARSEN et al., 2023). However, being limited to Java limits its applicability, as many projects are written in other programming languages. This motivates the need for structured merge tools that can work with multiple programming languages. Chapter 3 presents two novel tools that aim to fill this gap.

3 GENERIC STRUCTURED MERGE

To reduce the problems discussed in the previous chapter, and to improve merge accuracy for a wide range of programming languages, we propose LastMerge (*Language Agnostic Structured Tool for Code Merging*), a *generic* structured merge tool that can be easily configurable for each language. This chapter focuses on describing the design and implementation of such generic structured merge tools. Section 3.1 explains the main decisions of LastMerge which is built as a generic counterpart of *jDime* — adapting its algorithms to work on a generic tree. Section 3.2 presents a high level view of the ones chosen for *Mergiraf*, a recently proposed open-source tool that counterparts with *Spork*, that we use in our experiment to understand whether our evaluation results are specific to LastMerge or generalize beyond our particular design choices.

3.1 LASTMERGE

To achieve generality, LastMerge relies on a core merge engine that operates over generic trees produced by *Tree Sitter* (BRUNSFELD, 2025), an extensible and fast parser framework that has been instantiated for more than 350 languages, and is in production at GitHub. *Tree Sitter* allows users to define a Context-Free Grammar (CFG) using a domain-specific language (DSL) to generate a parser. This parser builds a Concrete Syntax Tree (CST), a tree representation of the source code that preserves all syntactic elements; nodes are represented as either *Terminal* (leaf nodes, such as literals) or *NonTerminal* (internal nodes, such as method declarations). Unlike Abstract Syntax Trees (ASTs), CSTs retain more granular and less abstract information, as illustrated in Figure 5.

Although several parser generators exist, the main advantage of *Tree Sitter* lies in its extensive collection of community maintained grammars for most programming languages used in industry. This ecosystem enables developers to build tools that aim to be language independent by focusing on a core that operates over generic tree nodes, while delegating the parsing to *Tree Sitter*. LastMerge employs the same strategy, abstracting away the language-specific aspects of structured merge that can be configured by the user through a simple interface. We discuss such aspects as well as the configuration interface of LastMerge on Chapter 5. Architecturally, LastMerge follows a sequential pipeline composed of three

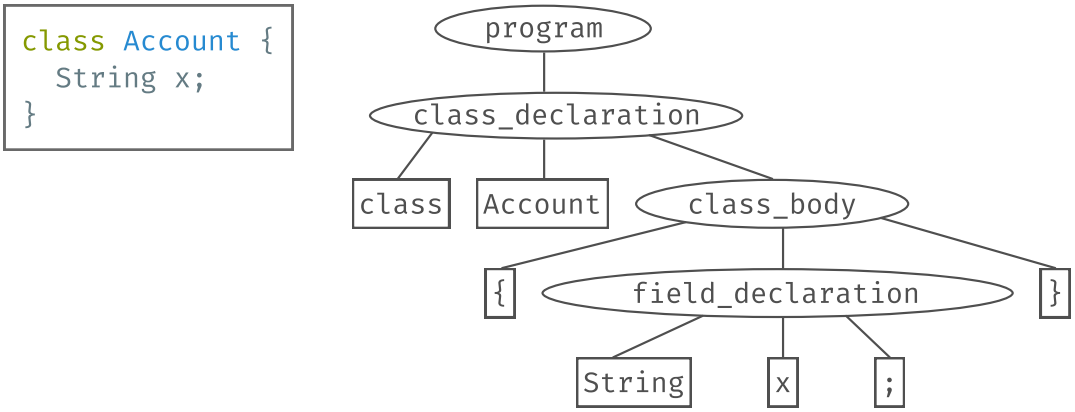

Figura 5 – An CST produced by *Tree Sitter*. An CST represents all the syntactic information of the original source code, including lexical elements.

main steps: *parsing*, *matching* and *merging*. Following sections describe these steps in detail, highlighting the design decisions and implementation details for each of them.

3.1.1 Parsing

LastMerge relies on *Tree Sitter* to convert each revision into a Concrete Syntax Tree (CST). In special, *NonTerminal* nodes can be marked as *Unordered*, indicating that their children can be safely permuted without affecting program semantics. Nodes may also include an optional identifier used to uniquely distinguish them among their siblings. Both identifier assignment and children ordering are language-specific aspects that must be properly configured to ensure correct behavior.

To better illustrate these concepts, consider the snippet of Java code in Figure 6. In this scenario, the order of the declarations within the class `Account`, such as the method `withdraw` and the field declaration `balance`, can be altered freely, as they are marked as unordered nodes, without affecting the program semantics. Additionally, the signature of the method `withdraw(int amount)` can be used to uniquely identify it among the children of the `Account` class.

```
public class Account {
  private int balance;

  public static void withdraw(int amount) {
    // ...
  }
}
```

Figura 6 – An example Java code for class Account

3.1.2 Matching

This and the next step are based on *jDime*'s algorithms (APEL; LEßENICH; LENGAUER, 2012), which we adapted to work in our generic language-independent tree. We choose *jDime* as the basis for LastMerge because, at the time of this work, it was the most mature and benchmarked implementation of a structured merge tool available in the literature. Matchings between each pair of revisions — (*base*, *left*), (*base*, *right*), and (*left*, *right*) — are first computed by associating nodes in each revision in a pair with corresponding nodes in the other revision. The matching process disallows pairing a *Terminal* node with a *NonTerminal* node. Moreover, even when nodes share the same type (*NonTerminal* or *Terminal*), they can only be matched if their kinds (`method declaration`, `literal`, etc.) are identical. These restrictions ensures the merge algorithm only matches semantically equivalent nodes. In the example of Figure 6, the method declaration `withdraw` can be matched only with another method declaration, and will never be matched with field declaration of `balance`.

The rules used to compute the matchings depends heavily on the types of nodes being compared. This decision is summarized in Algorithm 1. Matching *Terminal* nodes, for example, is relatively straightforward; a match is assigned only if their values are identical. This approach also applies to other literal types, such as integers or booleans. Matching *NonTerminal* nodes is more complex. When their root nodes match — either because they share the same identifier or kind — we recursively compute matches between each pair of children to find the maximum matching. This process occurs level-wise, so nodes only match with others siblings at the same tree level. Again, in Figure 6, the method declaration `withdraw` can only match with other sibling method declarations within the same class. Algorithms that can compare nodes across different levels exist and can be more precise as they can detect, for example, moved or renamed

code blocks. However, they are not used in LastMerge due to their higher computational cost — since such problems are known to be in NP-Hard.

```
   Input: A pair of nodes L and R.
   Output: The maximum number of matchings between L and R.
 1 if L and R have different kinds then
 2 |  return 0;
 3 end
 4 if L and R are Terminal nodes then
 5 |  if L and R have the same value then
 6 |  |  return 1;
 7 |  end
 8 |  else
 9 |  |  return 0;
10 |  end
11 end
12 if L and R are NonTerminal nodes then
13 |  if L and R are Unordered then
14 |  |  return unordered_tree_matching(L, R);
15 |  end
16 |  else
17 |  |  return ordered_tree_matching(L, R);
18 |  end
19 end
```

Algoritmo 1: Tree Matching

The specific algorithm to compute matchings among the children of two *NonTerminal* nodes depends on whether their children are ordered or unordered. Given two nodes with *ordered* children L and R, we compute their matchings using the procedure described in Algorithm 2. The algorithm is based on the original one proposed by Yang (1991), which generalizes the problem of computing the largest common subsequence from strings to trees. It employs a dynamic programming approach to find the number of matches (W) and the maximum matching (M) between two tree nodes L and R in quadratic time. In this context, an element $W[i][j]$ of the matrix W represents the maximum number of matches between the i-th child of L and the j-th child of R. On the other hand, an element $M[i][j]$ of the matrix M represents the maximum number of matchings between the first i children of L and the first j children of R. At the end of the algorithm, $M[m][n]$ contains the maximum number of matchings between the two nodes, where m and n are the number of children of L and R, respectively. Both matrixes M and W are populated by recursively traversing both trees and computing the maximum matching between each pair of their children (L_i, R_i).

If nodes have unordered children, in the more general case, we rely on a linear programming

Input: A pair of nodes L and R of the same kind.
Output: The maximum number of matchings between L and R.

```
1  m ← number of children of L;
2  n ← number of children of R;
3  M ← (m + 1) × (n + 1) ;                                  // Auxiliary matrix
4  for i ← 1..m do
5  |   for j ← 1..n do
6  |   |   W[i][j] ← calculate_matchings(L_i, R_j) ;          // Match children
7  |   |   M[i][j] ← max(M[i][j − 1], M[i − 1][j], M[i − 1][j − 1] + W[i][j]);
8  |   end
9  end
10 return M[m][n] + 1 ;              // We add 1 to account for the root nodes
```

Algoritmo 2: Ordered Tree Matching - Yang's algorithm

approach to find the maximum matching, similar to *jDime*'s original implementation (APEL; LEßENICH; LENGAUER, 2012). On this approach, computing the highest number of matches between the trees is equivalent to computing the maximum number of matches in a weighted bipartite graph. Such problem is also known as the *assignment problem*, and it can be solved algorithmically in *cubic* time, as shown in Algorithm 3. Given two trees L and R, the algorithm first computes the number of matches between each pair of children nodes (L_i, R_i) in the left and right trees. The auxiliary function `solve_assignment_problem` computes the maximum matching using the well known *Kuhn-Munkres* algorithm (KUHN, 1955), a linear programming inspired algorithm for solving the *assignment problem*. LastMerge uses the implementation of the *Kuhn-Munkres* algorithm provided by the `pathfinding` crate[1].

Input: A pair of nodes L and R of the same kind.
Output: The maximum number of matchings between L and R.

```
1  m ← number of children of L;
2  n ← number of children of R;
3  M ← m × n ;                          // Matrix to store children matchings
4  for i ← 0..m do
5  |   for j ← 0..n do
6  |   |   M[i][j] ← calculate_matchings(L_i, R_j) ;          // Match children
7  |   end
8  end
9  max_matches ← solve_assignment_problem(M) ; // Kuhn-Munkres algorithm
10 return max_matches + 1 ;          // We add 1 to account for the root nodes
```

Algoritmo 3: Unordered Tree Matching using the Assignment Problem

Despites its power, the *Kuhn-Munkres* algorithm is computationally expensive. To reduce computational cost, we can apply an optimization to the matching process when all children

[1] <https://docs.rs/pathfinding/latest/pathfinding/>

being matched have identifiers. That occurs because if a node in one revision has an identifier that matches a node in the other revision, we can assign a match without needing to compute the maximum matching. Conversely, if a node in one revision has an identifier that does not match any node in the other revision, we assume it has no match on the other revision. This optimization reduces the matching algorithm to simply traversing both sets of children and assigning a match when nodes have identical identifiers. This enhanced version can be implemented naively in quadratic time, although it could be further optimized to linear time by using a hash table to store the identifiers of the nodes in one revision, and then checking for matches in the other revision.

Note that in both the ordered and unordered cases, instead of calling themselves during the recursion, both algorithms delegates the call to the more general `calculate_matchings` function, presented in Algorithm 1. This is done to ensure the most appropriate algorithm is selected, depending on wether the nodes being compared are either *Terminal* or *NonTerminal* and, in the case of the later, wether their children are ordered or unordered.

3.1.3 Merging

LastMerge constructs the final merged tree using a depth-first strategy, starting from the pair of root nodes of each revision — and its common ancestor if existing, and recursively traverses the tree until all nodes are merged. It leverages the previously computed matching information to decide which nodes to retain or discard, how to integrate concurrent changes to the same node, and when to report conflicts. When merging *Terminal* nodes, LastMerge uses textual merge to compute the final version of the node, which is then added to the merged tree. When merging *NonTerminal* nodes, the algorithm needs to traverse the children of the nodes in each revision and apply the merge logic recursively. The logic however, differs wether the nodes are *Ordered* or *Unordered*.

In the case of *Ordered* nodes, the algorithm operates by traversing both sets of children in pairs, as shown in Algorithm 4. On this main loop, the algorithm first retrieves some indicators using the function `calculate_indicators`. It uses the matching information for the current children being processed from both revisions to retrieve their matching base nodes, their matching node on the opposite revision and wether they are matched with each other. Using this information, the algorithm infers the changes made in each revision, and assign them to operations that can be performed on the tree using the `choose_operations` function. Table

Tabela 1 – Merging cases for *Ordered* nodes. Here, both c_L and c_R represents the children of L and R being merged. l, r and b are placeholders for any (other) node from the left, base, and right trees. Each column represents a different indicator that can be used to determine the actions to be taken during the merge. A checkmark indicates that a match is present. The indicators $c_L \to r$, $c_L \to b$, $c_R \to l$ and $c_R \to b$ denote a match from c_L or c_R to any node in an opposing tree. The column $c_L \iff c_R$ indicates whether a bidirectional matching between the nodes was found. The remaining columns show the actions to be taken for the corresponding configuration of indicators. The last two columns indicate whether it is safe to advance the pointer for c_L and c_R. The table is based on Seibt et al. (2022) and adapted to the context of LastMerge.

$c_L \iff c_R$	$c_L \to r$	$c_L \to b$	$c_R \to l$	$c_R \to b$	Result	If left subtree changed	If right subtree changed.	If both subtrees changed.	Move L?	Move R?
✓	✓	✓	✓	✓	Merge(c_L , c_R)				✓	✓
✓	✓		✓		Merge(c_L , c_R)				✓	✓
	✓	✓		✓	Deletion(c_R)			Conflict(c_R, $none$)		✓
	✓	✓			Addition(c_R)					✓
	✓			✓	Deletion(c_R)			Conflict(c_R, $none$)		✓
	✓				Addition(c_R)					✓
		✓	✓	✓	Deletion(c_L)	Conflict(c_L , $none$)			✓	
		✓	✓		Deletion(c_L)	Conflict(c_L , $none$)			✓	
		✓		✓	Deletion(c_R), Deletion(c_L)	Conflict(c_L , $none$)	Conflict($none$, c_R)	Conflict(c_L , c_R)	✓	✓
		✓			Addition(c_R), Deletion(c_L)	Conflict(c_L , $none$)			✓	
			✓	✓	Addition(c_L)				✓	
			✓		Addition(c_L)				✓	
				✓	Addition(c_L), Deletion(c_R)		Conflict($none$, c_R)		✓	✓
					Conflict(c_L , c_R)				✓	✓

1, based on Seibt et al. (2022), summarizes the operations for each valid configuration.

Once the operations are determined, the algorithm applies them to the merged tree. For example, if a node from the left revision does not have a match in right, but has one in the base revision, the algorithm infers that the node was removed in the right revision. However, if the left node was modified in comparison to the base ancestor, the algorithm reports an *edit/delete* conflict. These operations are applied by the function `apply_all`. After the operations are applied, the pointers to the children of each tree might need to be updated. This is done by calling the `next` function on each set of children, which returns the next child in the list, or `null` if there are no more children to process. Wether to advance or not the pointers depends on the operations applied, as summarized on Table 1. This process continues until all children from both revisions are processed.

After one of the children lists is fully consumed, the algorithm proceeds to merge the remaining nodes on the other side of the merge. This is done by traversing the remaining children of the other revision and applying the same logic as the main loop. The algorithm finishes by returning a new node of the same kind as the original nodes, containing its merged children.

The merging of *Unordered* nodes follows a similar, but simpler, approach described in Algorithm 5. As the order of the children can be freely permuted, the algorithm does not need to traverse them in pairs. Instead, it starts by first processing the children of the left revision and them the ones from the right revision. For each child it processes, the algorithm

Input: A pair of *Ordered* nodes L and R of the same kind. The matching information previously calculated
Output: The final merged node.

```
1  result_children = Empty list;
2  cs_L = children of Left;
3  cs_R = children of Right;
4  cur_left = cs_l.next();
5  cur_right = cs_r.next();
6  while cur_left ∧ cur_right do
7  |  ind = calculate_indicators(cur_left, cur_right);
8  |  ops = choose_operations(ind) ;              // Operations from Table 1
9  |  (move_L, move_R) = apply_all(ops, cur_left, cur_right, result_children);
10 |  if move_L then
11 |  |  cur_left = cs_L.next();
12 |  end
13 |  if move_R then
14 |  |  cur_right = cs_R.next();
15 |  end
16 end
17 while ¬done_L do
18 |  ind = calculate_indicators(cur_left);
19 |  op = choose_operation(ind);
20 |  apply(op, cur_left, result_children);
21 |  cur_left, done_L = cs_L.next();
22 end
23 while ¬done_R do
24 |  ind = calculate_indicators(cur_right);
25 |  op = choose_operation(ind);
26 |  apply(op, cur_right, result_children);
27 |  cur_right, done_R = cs_R.next();
28 end
29 return new NonTerminalNode(kind, result_children);
```

Algoritmo 4: Merging Ordered Nodes

recovers its matching node in both the opposite revision and the base ancestor. It them uses this information to determine the resulting operation as follows. In case a node is added only by a single revision, it is simply added to the final tree. However, if the same node is added by both revisions, the algorithm recursively merges it. The same occurs for existing nodes that might have been modified by both revisions. The final case to be considered is the opposite parent removes the node. Here, the algorithms checks wether changes were made to this node, and if that is the case, it adds a special node to report a conflict.

Since the algorithm traverses both sets of children independently, it introduces the risk of processing the same node twice — the first in the left loop and the second in the right loop. To

guarantee that nodes are only processed once, the algorithm uses the set `processed_nodes` to keep track of the nodes that have already been processed. Thus, after applying each operation on the left loop, the algorithm adds the corresponding nodes into this set. Note that, which nodes have already been processed in each operation can vary. For example, when merged recursively, both the left node and its matching right node are added to the set. Then, when iterating through the children of the right revision, it simply skips the children that have already been processed.

.

Input: A pair of *Unordered* nodes L and R of the same kind. The matching information previously calculated.
Output: The final merged node.

```
1  result_children = Empty list;
2  processed_children = Empty set;
3  for cur_left ∈ children of L do
4      m_bl = matching base node for cur_left;
5      m_lr = matching right node for cur_left;
6      if ¬m_bl ∧ ¬m_lr // Node added only by left
7       then
8          result_children.push(cur_left);
9          processed_children.add(cur_left);
10     end
11     else if ¬m_bl ∧ m_lr // Node added by both left and right
12      then
13         result_children.push(merge(cur_left, m_lr));
14         processed_children.add(cur_left);
15         processed_children.add(m_lr);
16     end
17     else if m_bl ∧ m_lr // Node matched in base and right
18      then
19         result_children.push(merge(cur_left, m_lr));
20         processed_children.add(cur_left);
21         processed_children.add(m_lr);
22     end
23     else if m_bl ∧ ¬m_lr // Node removed by right
24      then
25         if cur_left was modified then
26             result_children.push(conflict(cur_left, none));
27         end
28         processed_children.add(cur_left);
29     end
30 end
31 for cur_right ∈ children of R do
32     if cur_right ∈ processed_children then
           // Skip already processed children
33         continue;
34     end
       // Analogous to the previous loop, but for right children
35 end
36 return new NonTerminalNode(kind, result_children);
```

Algoritmo 5: Merging Unordered Nodes

3.2 *MERGIRAF*

Similarly to LastMerge, *Mergiraf* is based on *Tree Sitter*'s parse infrastructure and language-independent tree. However, instead of adapting *jDime*'s algorithms to a generic context, *Mergiraf* opts for adapting *Spork*'s algorithms. By design, *Mergiraf* explores auto tuning (APEL; LEßENICH; LENGAUER, 2012), first attempting an unstructured merge of its input files. Only if conflicts arise during this attempt, it resort to structured merge.

When using structured merge, the tool builds fictional trees upon the conflicts found during the line-based merge of the file. This allows *Mergiraf* to aggressively pre-assign matchings between the revisions, which significantly speeds up the matching process. The remaining matchings between each pair of revisions are computed using only the GumTree algorithm (FALLERI et al., 2014), differently from LastMerge that combines different algorithms. Finally, revisions are merged into the final artifact using the same algorithm used by *Spork* (LARSEN et al., 2023).

4 EVALUATING GENERIC STRUCTURED MERGE

To understand the impact that a *generic* structured merge technique might have on merge accuracy and computational performance, we run an experiment with generic tools (which rely on matching and merging language independent trees) and their language specific counterparts. The counterpart influence of existing state of the art, language specific, tools on the design and implementation of the generic ones is reflected in our choice of tools. Thus, we choose to pair LastMerge with *jDime*, and *Mergiraf* with *Spork*, reflecting the influence of existing state of the art, language specific, tools on the design and implementation of the generic ones.

This pairing helps to isolate the generality aspect, mitigating bias that could arise from the effect of design decisions (algorithms, etc.) not related to implementing the generic requirement. This setting, especially with two generic tools, also helps us to investigate whether results are consistent across two state of the art algorithms for structured merge. We intentionally skip the pairings of LastMerge with *Mergiraf* and *jDime* with *Spork* from our study, as these pairings would not allow us to isolate and understand the impact of the generic requirement in the merge scenarios. We replay merge scenarios using each tool (the generic ones instantiated with Java syntactic and semantic details), and collect data on runtime, behavioral divergences, and merge accuracy.

In this chapter, Section 4.1 presents the research questions we address and how they helps us guide the experiment to meet our goals. Section 4.2 describes our sampling process, that uses a robust and state-of-the-art dataset for executing software merging experiments. Section 4.3 further describes our methodology by describing our experiment design. Section 4.4 presents the results of our experiment, and discussing how these results help us to answer our research questions. Finally, Section 4.5 discusses the threats to validity of our study, and how we mitigate them.

4.1 RESEARCH QUESTIONS

With the goal of understanding whether *generic* structured merge tools can effectively replace language-specific ones, achieving similar levels of accuracy and efficiency, we ask the following research questions:

4.1.1 RQ1: How *generic* structured merge impacts merge accuracy?

To address this question, we compute the number of spurious conflicts (false positives) and actual missed conflicts (false negatives), but we do that comparatively, only when the pair of tools being compared yield different results. So if both tools erroneously report a conflict, we do not consider that as a false positive in our analysis. We pay only attention to scenarios where one tool reports a conflict and the other does not, for instance. This is needed because establishing sound conflict ground truth for a large sample is hard. Thus, we focus our analysis on scenarios where the tools disagree on the presence of conflicts, as these are the scenarios that matter the most to developers. We perform a relative analysis that discards the scenarios where both tools of the pair agree on the presence or absence of conflicts, using the concepts *added false positives (aFPs)* and *added false negatives (aFNs)* (CAVALCANTI; BORBA; ACCIOLY, 2017; CAVALCANTI et al., 2024), which are explained in detail latter. To answer this question, we also manually analyze a number of cases to understand whether differences in conflict detection accuracy occur due to programming language independence or other factors.

4.1.2 RQ2: How *generic* structured merge impacts merge runtime performance?

To answer this research question, we measure the runtime of each tool in every merge scenario. Pairwise comparisons enable a clearer understanding of how different tree structures affect the performance of structured merge tools. We also examine whether programming language independence imposes a prohibitively high performance cost.

By answering these questions we hope to understand whether generic structured tools such as LastMerge have the potential to pave the way for broader adoption of structured merge in industry, as they can be easily adapted to multiple languages.

4.2 SAMPLING

For comparing the merge tools, we use the dataset of merge scenarios published by Schesch et al. (SCHESCH et al., 2024). The sample consists of 5,983 merge scenarios from 1,116 open source projects. Projects are extracted from GitHub's Greatest Hits (GITHUB, 2020) and Reaper (MUNAIAH et al., 2017) datasets, and were carefully filtered so that users can rely on significant buildable Java projects that are relevant within the open-source community. The

dataset includes scenarios with non-trivial test suites that pass on both parents within a specified timeout. This is particularly useful because it allows us to rely on test execution to check merge accuracy; if tools yield different results but project tests pass in the results of just one of the tools, we know the other tool has a problem.

When trying to replicate the original dataset, a number of scenarios could not be retrieved due to external factors— such as when the GitHub repository is unavailable. We discard these in our study. We further filter the sample to ensure each scenario contains at least one file that was mutually modified by both parents. This excludes scenarios where merging is trivially achieved by selecting the revision that introduces the changes; the tools would yield the same results for these scenarios, not contributing to our comparative analysis. Additionally, we remove scenarios in which any of the evaluated merge tools crashed during execution. As a consequence of these filtering, we perform our experiment on a sample of 5,229 scenarios, spanning 13,675 mutually modified files.

4.3 CHECKING MERGE ACCURACY AND PERFORMANCE

To answer RQ1, following the merge process, we collect metrics to estimate merge accuracy, as explained in Section 4.1. In particular, we adopt a relative comparison (CAVALCANTI; BORBA; ACCIOLY, 2017; CAVALCANTI et al., 2024), computing the occurrence of false positives and false negatives of one tool *in addition* to the other tool in the same pair, as shown in Figure 7. Given a pair of tools (A, B), we say tool A suffers an *added false positive (aFP)* in a scenario if it reports an spurious conflict that tool B does not. Similarly, A suffers an *added false negative (aFN)* in a scenario if it fails to report an actual conflict detected by B.

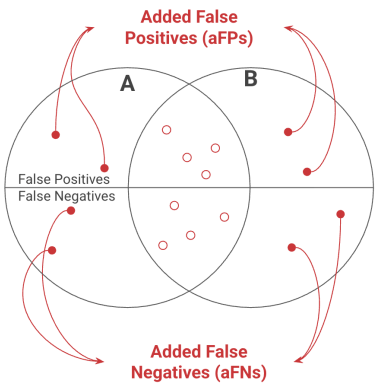

Figura 7 – A Venn diagram illustrating the concepts of added false positives (*aFPs*) and added false negatives (*aFNs*) for a pair of tools (A, B). Note that scenarios in which both tools incorrectly report or miss conflicts are not considered in our analysis.

Note that our analysis metrics assumes both tools disagree on the presence of conflicts in a scenario. This aligns with our goal to compare tools relatively, focusing not on the absolute number of conflicts each tool reports but on how they differ. Remember we are particularly interested in understanding whether the *generic* nature of a tool impacts its accuracy in comparison to its language-specific counterpart. This way, we intentionally discard scenarios where both tools incorrectly identify or miss conflicts, as these scenarios do not help us understand the relative performance of the generic requirement. Moreover, scenarios with disagreement are critical for developers, as incorrect conflict detection can result in faulty merges that are costly to fix. We calculate the number of *aFPs* and *aFNs* for a pair of tools (A, B) by combining syntactic and semantic approaches, as illustrated in Figure 8.

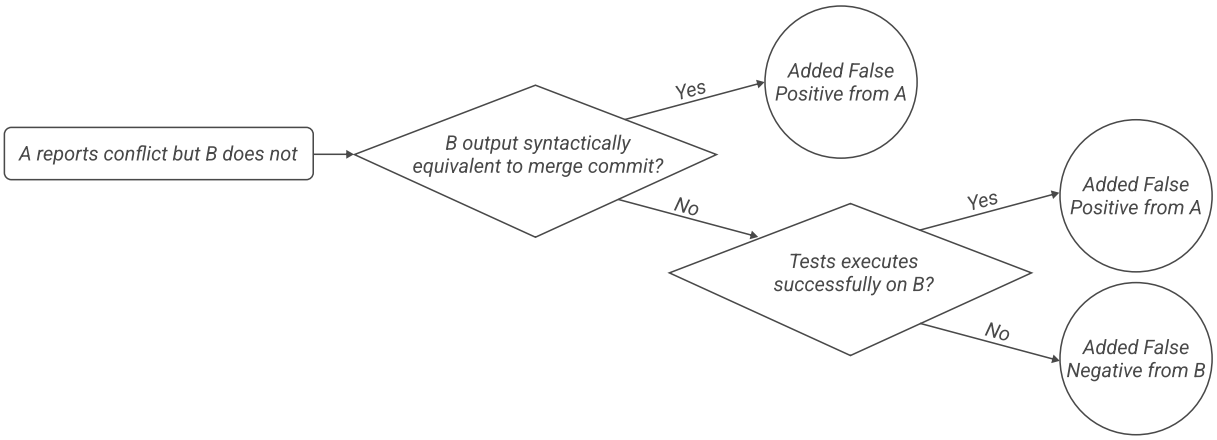

Figura 8 – Deciding whether a tool has an added false positive (*aFP*) or an added false negative (*aFN*) in a merge scenario (merge commit in the repository, its parents, and their common ancestor). The output of B is first compared to the merge commit in the repository. In summary, in the scenario that A reports a conflict that B does not, we check whether B successfully and accurately resolves these conflicts.

Given a scenario where tool A reports a conflict and B does not, we aim to determine whether B has successfully and accurately resolved the conflict. We begin by checking if the output produced by B is syntactically equivalent to the merge commit of the original project repository, as developers are presumably happy with such result. One caveat of this approach is that both *jDime* and *Spork* often modify the original source code during pretty-printing by inserting or removing tokens. This happens because these tools use more abstract tree representations of the source code, which do not retain all the syntactic details of the original revisions. We mitigate this issue by normalizing the formatting of the output files before performing the syntactic equivalence check. This normalization is performed by running the

pretty-printer of the tool in the files that will be checked, so that formatting is consistent across all files. We then execute LastMerge to parse the files and execute its matching algorithms to check whether the two resulting trees fully match, which indicates syntactic equivalence. If the files are found to be syntactically equivalent, we assume B correctly resolved the conflict reported by A, and classify the scenario as an *aFP* for A.

Conversely, if B's output is not syntactically equivalent, we rely on executing the project test suite on the output generated by B. If all tests pass, we assume that the changes introduced by both parents are non-conflicting and that B successfully integrated them. Since A reported a conflict and thus failed to integrate the changes, we classify this as an *aFP* for A. However, if the tests fail, we assume B did not correctly detect the conflict that A detected, thus resulting in an *aFN* for B.

Finally, as a last step to answer RQ1, we conduct a manual code analysis to better understand the reasons behind tool differences. We randomly select 5 scenarios with *aFPs* and 5 with *aFNs* for each tool, resulting in a total of 40 scenarios. For each scenario, we investigate whether the observed differences in the tools outputs are due to programming language independence or from other factors, such as configuration differences or implementation details.

To answer RQ2, we collect runtime execution metrics by measuring the time each tool takes to merge each scenario. To do so, we aggregate the execution times across all files within a scenario. To minimize the influence of external factors, each tool is executed sequentially ten times per file, with runtime measured in each run. We discard the first measurement— used as a warm-up— and compute the average of the remaining nine runs to determine the runtime for each file.

We provide the scripts and data associated with this study in our online appendix.[1]

4.4 RESULTS AND DISCUSSION

In this section we present and discuss our findings, structured according to the research questions outlined in Section 4.1, and the two pairs of $(specific, generic)$ structured merge tools compared: (*jDime*, LastMerge) and (*Spork*, *Mergiraf*).

[1] <https://anonymous.4open.science/r/experiment-last-merge-3908>

4.4.1 RQ1: How *generic* structured merge impacts merge accuracy?

As explained in Section 4.3, our analysis is comparative. For each tool pair, our analysis focuses on scenarios in which the tools disagree on the existence of conflicts, as we are not concerned with cases where both tools fail or where both successfully perform the merge. So we first summarize in Table 2 the agreement and disagreement on the existence of conflicts between each pair of merge tools. We observe a disagreement rate on the existence of conflicts of 7.53% between LastMerge and *jDime*, and of 12.22% between *Mergiraf* and *Spork*. So, based on our sample, we observe the *generic* structured merge tools behaving differently than their language-specific counterparts in a minor, but considerable, part of cases.

Tabela 2 – Agreement rate on conflict existence in the analyzed merge scenarios. The total sum in each column can vary because not all scenarios were successfully integrated by each tool.

Situation	***jDime* vs LastMerge**	***Spork* vs *Mergiraf***
Agreement on existence of conflicts	4909 (92.5%)	4697 (88.7%)
Disagreement on existence of conflicts	400 (7.5%)	601 (11.3%)

In the following, we discuss how these behavior differences impact the accuracy of the tools to detect and resolve conflicts. Furthermore, we examine whether these differences are related to the generic aspects (language independent trees and algorithms) of both LastMerge and *Mergiraf*. We do this first for the pair LastMerge and *jDime*, and then for the pair *Mergiraf* and *Spork*.

LastMerge *and jDime*

Table 3 shows the results of the analysis of the accuracy of LastMerge and *jDime* in terms of *aFPs* and *aFNs*. The results indicate that LastMerge reports fewer *aFPs*, but exhibits nearly three times more *aFNs* than *jDime*. In absolute terms, this corresponds to 56 extra *aFNs*, which is considerable. Proportionally to the number of scenarios analyzed, or even to the number of scenarios in which the tools differ, we observe less significant numbers (1.1% and 14%, respectively), which are nevertheless further considered for our manual analysis.

Tabela 3 – Comparison of added false positives (*aFPs*) and added false negatives (*aFNs*) between LastMerge and *jDime*.

Situation	*jDime*	LastMerge
Added false positives (*aFPs*)	153	130
Added false negatives (*aFNs*)	29	85

We conduct a statical test to assess the significance of the differences observed. We apply McNemar's test (MCNEMAR, 1947), takes as input the number of scenarios where each tool produced *aFPs* and *aFNs*, and evaluates the null hypothesis that both tools have the same error rate. This test is appropriate for our analysis as it evaluates the differences in performance between two classifiers (in this case, merge tools) on paired nominal data (the presence or absence of conflicts). The mathematical equation of the test is given by the formula $\chi^2 = \frac{(|b-c|-1)^2}{b+c}$, where b is the number of scenarios where LastMerge yielded either an *aFP* or an *aFN*. Conversely, c is the number of scenarios where *jDime* produced either an *aFP* or an *aFN*. In our case, $b = 85 + 130 = 215$ and $c = 29 + 153 = 182$. Yielding a p-value of 0.11, the test indicates that the differences in conflict detection accuracy between LastMerge and *jDime* are not statistically significant at the conventional level of 0.05.

As explained in more detail in the rest of the section, our manual analysis suggests that the observed discrepancies are primarily due to implementation details and configuration differences, rather than to LastMerge relying on language independent trees and algorithms. Although LastMerge algorithms borrow from *jDime* exactly to reduce confounding factors, they rely on substantially different trees, and minor differences in algorithm implementations are expected when involving different developers and programming languages, as in this case. More important, configuration details, especially in the Java configuration of structured tools, can lead to differences that can easily eliminated by adjusting the configuration.

Starting with false positives, we found that in *jDime* three out of five analyzed scenarios occur due to inadequate tool configuration. To illustrate this, consider the example in Figure 9. Starting from the *Base* revision, *Left* adds the field declaration code, while *Right* independently adds the field declaration age. Despite these changes not being conflicting, *jDime* incorrectly reports an *insert/insert* conflict. This occurs because, during matching, *jDime* does not assign unique identifiers to field declarations and relies solely on its structural matching algorithm, which assigns a partial matching between the two properties— as they have the same node kind. In contrast, LastMerge properly treats the field name as a unique identifier and never matches nodes with different identifiers, thus correctly identifying the additions as non-

conflicting, resulting on a clean merge. *jDime*'s configuration could, and should, be adjusted to avoid this issue; it's not a fundamental limitation of the tool, or one that is inherent to language-specific tools.

Similar configuration differences also cause LastMerge to report false positives due to method renaming. For example, in Figure 9, *Left* modifies only the implementation of the method `greet`, while *Right* changes both its body and signature by adding an argument. Since LastMerge matches method declarations only when their signatures (name and argument types) are identical, it fails to match the different versions of `greet` from *Left* and *Right*. Instead, it mistakenly interprets *Right*'s change as the addition of a new method `greet(String greet)` and the removal of the original `greet()`. Furthermore, because LastMerge detects that *Left* also modifies the original method, it classifies this as a *modify/delete* conflict. In contrast, *jDime* matches methods by name and subtree structure, correctly identifying the correspondence between the renamed methods and producing a clean merge. Renaming conflicts are a common source of *aFPs* in structured merge tools (CAVALCANTI; BORBA; ACCIOLY, 2017; LEßENICH et al., 2017). Approaches to address this issue typically involve modifying general aspects of the matching process, which could be replicated in LastMerge, without prejudicing language independence.

Turning to false negatives, the conflicts missed exclusively by *jDime* (*aFNs*) also arise from differences in matching configuration. To illustrate, consider the example in Figure 10. The scenario involves independent changes to different constructors of the class *AbstractSolver*. *Left* retains only the no-argument constructor `AbstractSolver()`, modifies its body, and removes the field declaration seed. Meanwhile, *Right* keeps the constructor `AbstractSolver(long seed)` and modifies its body by adding a new logging statement. These are conflicting changes, as each parent modifies constructors that were removed in the other revision. Since *jDime* matches constructors based only on their name— not their full signature— it incorrectly matches the different constructors of *Left* and *Right*, merging both without reporting conflicts. However, due to the removal of the field declaration seed, the generated file cannot be compiled due to a missing symbol error. In contrast, LastMerge treats each constructor as distinct nodes, correctly detecting and reporting the conflict. jDime could be configured to match constructors by their full signature instead, which would result in the same behavior observed on LastMerge; once again this highlights that the differences in output observed arise from different configurations rather than language specific concerns.

We observe that *jDime* misses actual conflicts when merging generic type arguments, resul-

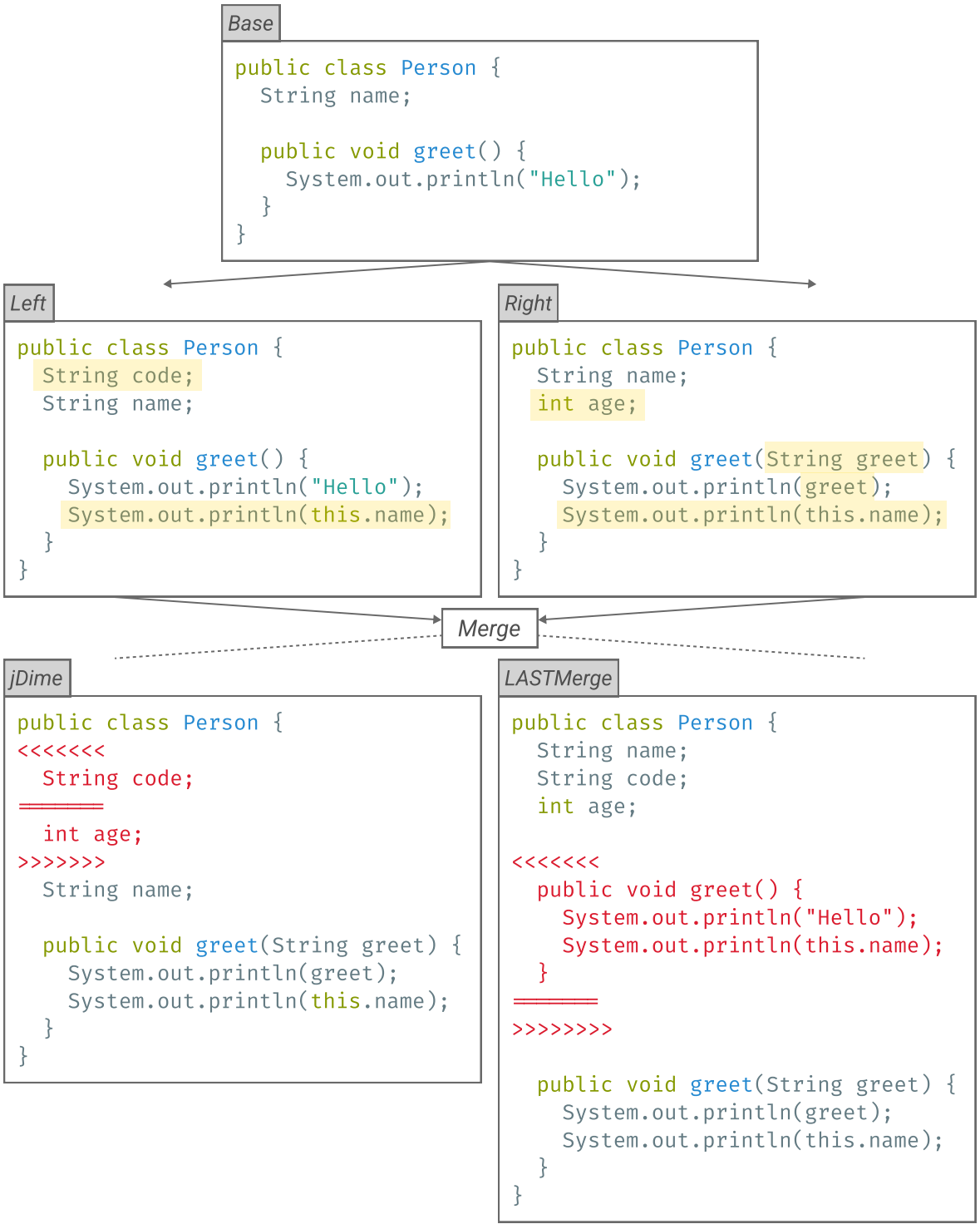

Figura 9 – Merge scenario illustrating the differences in conflict detection between LastMerge and *jDime*. Changes are highlighted in yellow.

Figura 10 – Merge scenario illustrating changes that lead into a false negative in *jDime*. Changes are highlighted in yellow.

ting in compilation errors. Similarly, LastMerge produces compilation errors when integrating throws declarations in method signatures. Both issues arise from incorrect configuration of children node ordering: *jDime* treats generic type arguments as unordered, while LastMerge treats throws declarations as ordered. Finally, such differences in configuration can be adjusted, and do not reflect fundamental limitations of the tools. Indeed, such configuration differences are not related to the language independence of LastMerge in any sense.

> *jDime* and LastMerge disagree on the existence of conflicts in 7.53% of the scenarios. A McNemar test yields a p-value of 0.11, suggesting that the differences in accuracy are not statistically significant. LastMerge has fewer *aFPs* than *jDime*, but exhibits nearly three times more *aFNs*. Our manual analysis shows that these discrepancies occur mainly due to implementation details and configuration differences, rather than to the language independent aspects of LastMerge.

Mergiraf and Spork

Table 4 shows that *Mergiraf* reports almost twice as many *aFPs* as *Spork*. Conversely, *Mergiraf* produces significantly fewer *aFNs*. Contrasting with the previous section, here the generic tool has more *aFPs* but fewer *aFNs*. Again, we conduct a statistical test to assess the significance of the differences observed. When applying the McNemar's test with values $b = 290 + 62 = 352$ and $c = 150 + 107 = 257$, we achieve a p-value of 0.0001. This time, differently from the previous result, it indicates that the differences in conflict detection accuracy between *Mergiraf* and *Spork* are statistically significant at the conventional level of

0.05. However, similarly to the previous section, we observe that the differences are mostly not due to the language independent aspects of the generic tool.

Tabela 4 – Comparison of added false positives (*aFPs*) and added false negatives (*aFNs*) between *Mergiraf* and *Spork*.

Situation	***Spork***	***Mergiraf***
Added false positives (*aFPs*)	150	290
Added false negatives (*aFNs*)	107	62

Our manual analysis indicates the design decision of *Mergiraf* to use auto-tuning strongly influences the differences observed between the tools. With this strategy, *Mergiraf* first attempts an unstructured merge of the revisions and falls back to a structured approach only when conflicts arise. In contrast, *Spork* always applies a structured merge algorithm. We observe that, in 4 out of 5 scenarios analyzed, *Spork* failed to reproduce clean merges that were previously achieved by *Mergiraf* through unstructured merge. These failures typically result from incorrect or missing node matchings, which lead *Spork* to report spurious conflicts (*aFPs*) in cases where *Mergiraf* produces conflict free results. Adding auto-tunnig to *Spork* would be trivial, though; conversely, modifying *Mergiraf* to skip auto-tunning can also be easily achieved.

Algorithmic differences also contribute to the discrepancies observed between *Mergiraf* and *Spork*. To illustrate this, consider the situation in Figure 11. *Left* modifies the type of the field declaration `timeElapsed`, while *Right* removes its declaration. This situation, where one revision deletes a node modified by the other, characterizes a *delete/edit* conflict. *Mergiraf* and *Spork* use the same merge algorithm, whose original implementation is not able to detect such conflicts (LARSEN et al., 2023). Instead, it silently deletes the node without reporting a conflict. This way, *Spork* considers only the removal of the field declaration `timeElapsed` by right, and completely ignores the changes made by *Left* to the same node. However, as *Left* also adds a new reference to `timeElapsed` in the `main` method, the resulting file fails to compile due to a missing symbol error. In contrast, *Mergiraf* extends the algorithm to keep track of deletions during the reconstruction of the merged tree. After the merged tree is fully constructed, it checks wether one of the deleted nodes was modified on the other revision, enabling it to correctly detect and report the conflict. Such extension is not a fundamental change in the algorithm, but rather an improvement to the original algorithm, which could be applied to *Spork* as well.

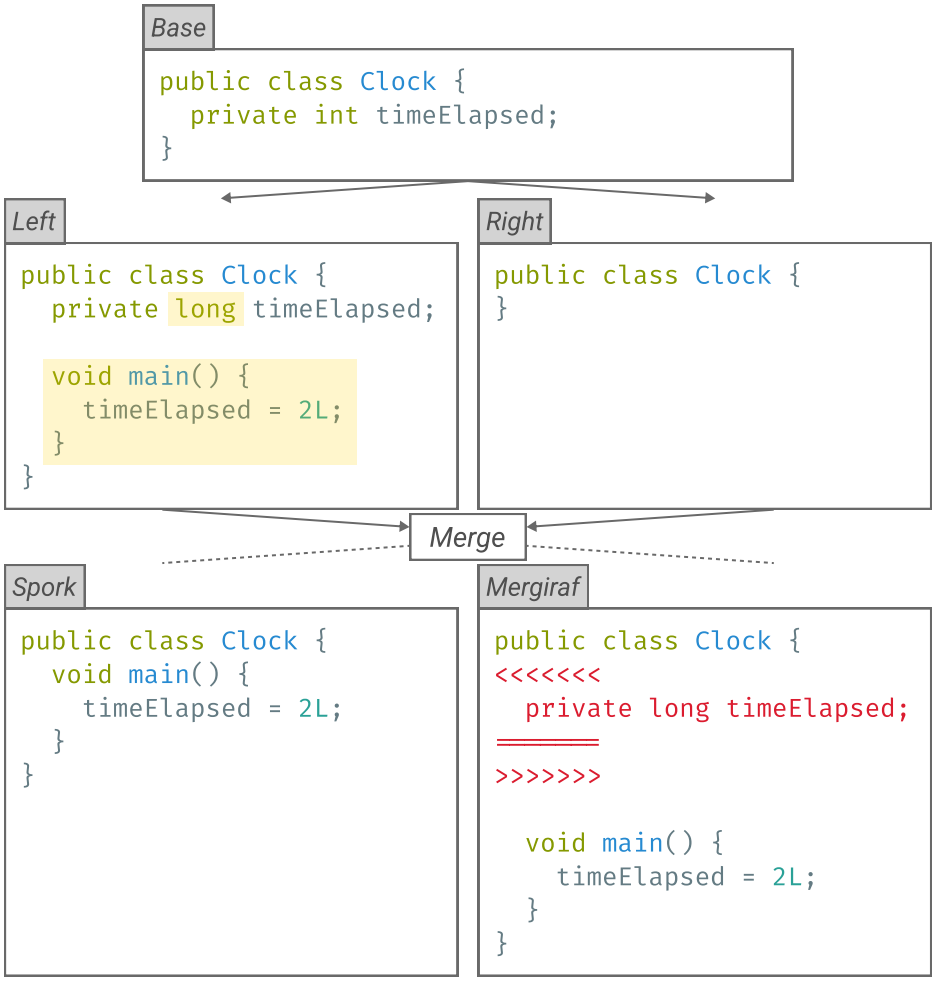

Figura 11 – A merge scenario illustrating a *delete/edit* conflict not detected by *Spork*. Changes are highlighted in yellow.

One can notice, however, that if *Left* did not introduce a new reference to the `timeElapsed` property, the file produced by *Spork* would remain semantically valid and compile without errors. In general, *delete/edit* conflicts only lead to issues when the changes introduced interfere semantically, thus leading into build-time errors (SILVA; BORBA; PIRES, 2022). Therefore, while *Mergiraf* adopts a more conservative strategy by always reporting *delete/edit* conflicts, this choice may result in a higher number of *aFPs*, as the scenarios where such conflicts truly affect the correctness of the merged program are relatively specific.

Spork and *Mergiraf* disagree on the existence of conflicts in 12.22% of the scenarios. *Mergiraf* has nearly twice as many *aFPs* than *Spork*, but exhibits significantly fewer *aFNs*. A McNemar's test yields a p-value of 0.0001, suggesting that the differences in accuracy are statistically significant. Our manual analysis shows that these discrepancies

occur mainly due to differences on the usage of auto-tuning and *Mergiraf* improvements to the original *Spork* algorithm, rather than the language independent aspects of the generic tool.

4.4.2 RQ2: How *generic* structured merge impacts merge runtime performance?

Figure 12 presents the runtime performance of the tools analyzed by merging the scenarios in our dataset. Overall, both LastMerge and *Mergiraf* outperform *jDime* and *Spork*, achieving speedups of at least one order of magnitude on average. We conduct a statistical analysis to assess the significance of the performance differences observed. We apply a t-test for independent samples, which compares the means of two independent groups. The test is conducted comparing the average runtimes of LastMerge and *jDime*, and then *Mergiraf* and *Spork*. For the test, we use the relevant runtime values of each tool per scenario, stated in Table 5. The t-test comparing LastMerge and *jDime* yields a p-value of 8.82e-54, while the test comparing *Mergiraf* and *Spork* yields a p-value of 1.743e-178. These results clearly indicate that the differences in runtime performance between both pairs of tools are statistically significant. These differences, however, arise mainly from implementation and design choices rather than fundamental algorithmic improvements or the language independent aspects of the generic tools.

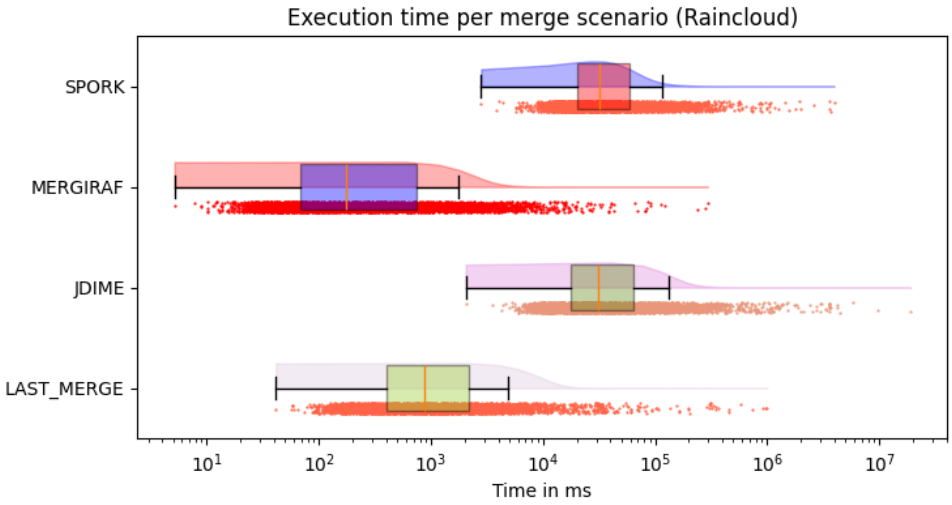

Figura 12 – A raincloud plot displaying runtime execution per merge scenario for each tool. Each dot represents the average of 9 sequential executions of each tool in the same merge scenario. Time is in logarithmic scale.

Tabela 5 – Average runtime and Standard Deviation (in seconds) per merge scenario for each tool.

Metrics (per scenario, in seconds)	*jDime*	LastMerge	*Spork*	*Mergiraf*
Average runtime	89.568	4.465	60.363	1.301
Standard Deviation	395.831	32.389	145.142	8.014

Both LastMerge and *Mergiraf* are implemented in Rust, whereas *Spork* and *jDime* use Java. Rust is a systems programming language that compiles directly to machine code, enabling efficient execution. In contrast, Java compiles to to an intermediate representation (bytecode) that is later interpreted and executed on the Java Virtual Machine. Furthermore, while Java achieves memory safety in runtime by using garbage collection, Rust takes a unique ownership model that enforces memory safety in compile time, eliminating runtime overhead. Since structured merge tools perform CPU-intensive algorithms and numerous in-memory, large tree manipulation operations, their runtime performance is highly sensitive to the characteristics of the implementation language.

Beyond the choice of programming language, other factors also influence performance differences. For example, *Mergiraf* employs techniques known to enhance efficiency, such as auto-tuning (APEL; LEßENICH; LENGAUER, 2012). In this approach, the tool first attempts an unstructured merge and resorts to structured methods only if conflicts arise. Invoking structured merge selectively leads to significant performance gains (SEIBT et al., 2022). Notably, an auto-tuning strategy could also be implemented without further effort, potentially yielding similar benefits for the other tools analyzed.

Finally, even if we focus our comparison on LastMerge— which does not use auto-tuning— and assume a conservative 10x performance penalty when comparing Java to Rust implementations, based on prior work (PEREIRA et al., 2017), LastMerge still delivers performance comparable to state of the art tools, integrating 80.2% of the scenarios in less than three seconds. This suggests that generic structured merge does not impose a computational cost that is prohibitive compared to existing tools, and can be used in most situations in industry.

4.5 THREATS TO VALIDITY

4.5.1 Internal validity

Our *aFPs* and *aFNs* analysis (see Section 4.3) relies on a heuristic that combines static and semantic analysis to approximate the existence of conflicts. Using the merge commit in the repository is a good approximation of the expected merge result, but developers might, for instance, have accepted the commit and immediately after noted a problem, later fixed it in a subsequent commit. Failing builds are a quite robust approximation of problem in the result yielded by the merge tool, but test is as robust as the project test suite itself. Nevertheless, this criteria has been used in recent work (SCHESCH et al., 2024) and is stronger than the ones used in previous work (ZHU; HE, 2018; ZHU; HE; YU, 2019; CAVALCANTI; BORBA; ACCIOLY, 2017), which focus only on the first part of our criteria.

Relying on manual analysis for identifying the reasons for false positives and false negatives can be challenging. Accurately classifying such scenarios require a deep understanding of the project implementation and even the original developers may sometimes overlook conflicting changes. Instead, our manual analysis focuses on understanding the underlying reasons for the differences in results produced by the tools. More specially, we focus on understanding wether these differences arise because of the introduction of language independence aspects in the generic tools. This approach is less error-prone, as the authors possess in-depth knowledge of the design and behavior of each tool. Additionally, all findings were thoroughly discussed among the authors to ensure accurate interpretations.

4.5.2 External validity

Our sample represents only a subset of possible merge scenarios. Specifically, we focus on cases extracted from publicly available Java projects on GitHub, which may not capture the full spectrum of programming practices. To mitigate this limitation, we rely on a comprehensive dataset introduced by the work of Schesch et al. (SCHESCH et al., 2024), which includes a diverse collection of projects spanning various domains and development practices, and can be considered the state of the art dataset for merge tool studies.

Finally, despite the language independence of LastMerge and *Mergiraf*, our findings may still be specific to Java, the programming language used in our evaluation. Since *jDime*

and *Spork* support only Java, generalizing our results to other languages would need structured merge tools for other languages, but these are hardly available. Additional studies comparing LastMerge and *Mergiraf* with language specific structured merge tools across different languages are necessary to better assess their applicability in broader contexts.

5 CONFIGURING LASTMERGE FOR DIFFERENT LANGUAGES

In the previous chapter, we demonstrated that *generic* structured tools instantiated for usage with Java can achieve comparable accuracy to state of the art Java-specific ones, while also not being computationally prohibitive. This is a promising result, as it suggests that LastMerge can be used in practice. However, this alone might not be enough to justify its adoption in industry, where the usage of different programming languages is common. In order to be useful in practice, LastMerge should be able to be configurable for usage with different programming languages, so that it can be used in a wide range of software projects.

Furthermore, configuring LastMerge for usage with a new programming language should be a low-effort process that requires little knowledge of the tool's internals. This allows users to quickly implement configurations for a broad range of programming languages and adapt it to their needs. In fact, if the configuration effort is too high, users might be discouraged from using it in practice, regardless of their advantages over different merge tools and strategies.

This chapter explores how LastMerge can be configured to work with different programming languages as well as assesses the effort involved in instantiating it for a new language. Section 5.1 explores the rationale behind the definition of such interface, by describing the language specific aspects that are relevant for structured merge. Section 5.2 discusses the effort involved in instantiating LastMerge, by building a minimal configuration for usage with C#. Section 5.3 presents known current limitations of LastMerge that limits its generalization power for usage with a few programming languages. Section 5.4 summarizes and concludes the discussion presented in this chapter.

5.1 DEFINING THE CONFIGURATION INTERFACE

In order to define the configuration interface of LastMerge, we first need to identify which aspects of structured merge are language dependent. Once we have identified these aspects, we can define an interface to abstract such aspects from the core of the tool. This way, an user interested in using LastMerge with a new programming language can implement the interface, without the need to modify the core of the tool.

5.1.1 Source code parsing

One of the most fundamental aspects of structured merge is the conversion of the source code into a tree like representation. This is important because structured merge algorithms operate over trees, rather than over raw text. This conversion is done by a parser, which is a program that reads the source code and outputs a tree representation of it. However, parsers are often language dependent, as each programming language has its own syntax and grammar.

LastMerge leverages *Tree Sitter* infrastructure to parse the source code. *Tree Sitter* is a parser generator that can be used to define parsers for different programming languages through a common DSL. It also provides a set of APIs to use such grammars to parse source code into Concrete Syntax Trees (CSTs). This way, a user interested in using LastMerge with a new programming language would need to provide a *Tree Sitter* grammar for the language. Fortunately, *Tree Sitter* already has grammars available for most programming languages used in industry[1], so the effort required to provide the grammar is often minimal.

5.1.2 Stopping compilation at an intermediate level

In a few contexts, it might be desirable to stop the compilation process at an intermediate level, rather than generating a full CST. This is the case, for example, when using *semistructured merge*. This merge strategy is based on the idea of combining both structured and unstructured merge, by applying structured merge to the higher level structures of the source code, such as classes and methods, while applying unstructured merge to the lower level structures, such as statements and expressions.

Thus, this mechanism can be simulated by stopping the compilation process at an intermediate level, such as the level of a method body. This way, the parsing algorithm would generate a tree that contains only the higher level structures, such as classes and methods, while leaving the lower level structures and groups the statements and expressions on a single *Terminal* node — which are merged using unstructured merge.

It is easy to see that this is also a language dependent aspect, as the intermediate level at which the compilation should stop may vary across programming languages. In LastMerge, the user can specify the node kinds at which the compilation should stop, by providing a list

[1] <https://github.com/tree-sitter/tree-sitter/wiki/List-of-parsers>

of node kinds that should be considered as *Terminal* nodes. For example, in Java, the tool could be configured to stop at the level of `method_body` nodes.

5.1.3 Parsing handlers

Although the pre-existing *Tree Sitter* grammars heavily facilitate extending LastMerge, its resulting trees might not always be the best representation for structured merge. To illustrate this, consider the scenario in Figure 13. In this scenario, the *Tree Sitter* grammar for Java places import declaration nodes directly under the root program node. This restricts the merge algorithm to reorder import declarations, as naively reordering program node children could result in a broken program — for example, one in which an import declaration ends up appearing after a class declaration.

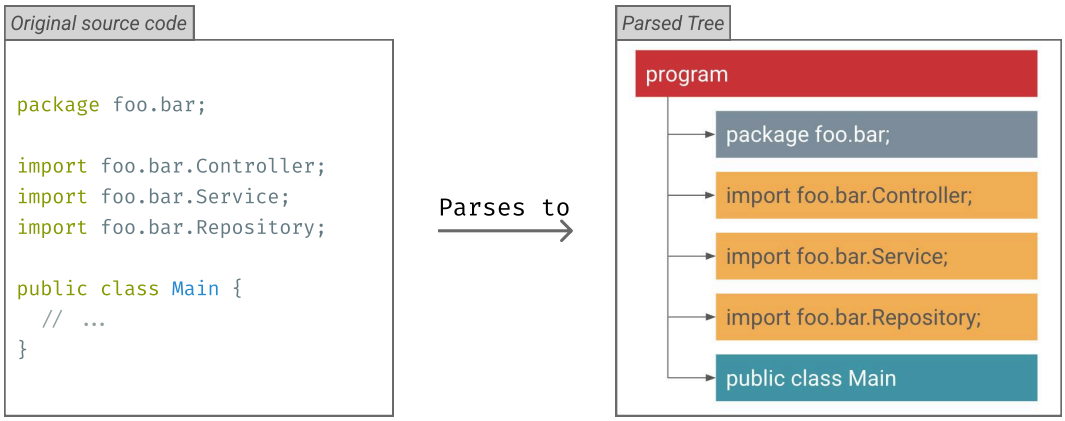

Figura 13 – CST generated by the Java *Tree Sitter* grammar for a program with import declarations. Note that import declarations are placed directly under the root node.

To address this issue, one could define an algorithm that transforms the CST generated by *Tree Sitter* into a more suitable representation for structured merge. For example, such handler could group the import declarations into a dedicated node, which would allow the merge algorithm to reorder them freely without breaking the program. Figure 14 Executing such post-processing algorithms after certain steps of the merge process is something explored by other merge tools (CAVALCANTI; BORBA; ACCIOLY, 2017), and such procedures are usually referred as *handlers*. Since, in the case of LastMerge, these *handlers* hook in after the parsing step is completed, we name them *parsing handlers*. From a code point of view, these handlers are functions that follows the signature `(root: CSTNode) -> CSTNode`, where `root` is the original root of the three. Such function them return a new root node, which incorpores

the transformations applied to the original tree. One can easily note that the implementation of *parsing handlers* are also language dependent, as the transformations applied to the trees might vary across the grammars and languages involved.

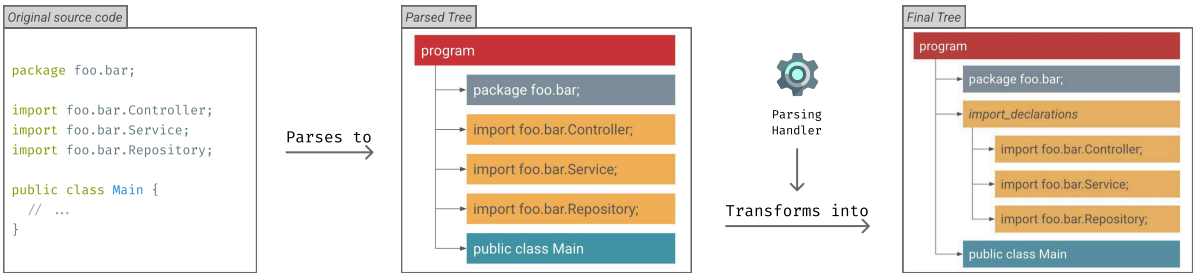

Figura 14 – The pipeline of parsing the original tree using Parsing Handlers. After the parsing is completed, LastMerge calls the registered parsing handlers to transform the tree. Here, the import declarations are grouped into a dedicated *NonTerminal* node, allowing the merge algorithm to reorder the imports freely without breaking the program.

5.1.4 Node children ordering

As shown in Chapter 2, structured merge is able to handle merge scenarios for Java in which both developers add different method declarations to a class within the same text area. Differently from unstructured tools, which would yield conflicts in this scenario, structured tools achieves a clean merge by reordering the method declarations — by simply juxtaposing them. This is possible because structured merge algorithms are able to reason about the structure of the code, rather than just the text.

However, wether a node can be reordered or not is a language dependent aspect. For example, in Java, enum variants can be reordered without prejudice to program semantics, which is not the case in Typescript.[2] Furthermore, similar structures in different programming languages may have their nodes named differently on the *Tree Sitter* grammar. For example, in the Java grammar, declarations within a class are grouped in a node named `class_body`, while in the C# grammar, such declarations are grouped within a `declaration_list` node. This way, a list of node names whose children can be reordered by the merge algorithm must be provided through the configuration interface.

[2] In Java, enum variants are converted into plain String representations. In Typescript however, such variants are numbered in order, starting from 1 by default.

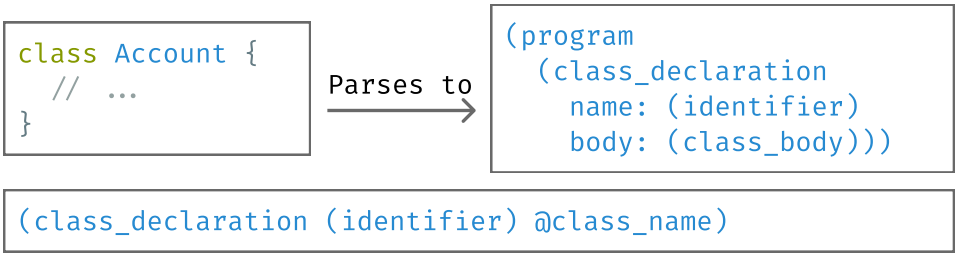

Figura 15 – On top a class declaration in Java and its *Tree Sitter* parsed version. On the bottom, a *Tree Sitter* query. Queries consist of one or more *patterns* that are specified as *S-Expressions*. In this example, the named capture *class_name* holds the identifier (name) of the class `Account`.

5.1.5 Node identifier extraction

Structured merge often assign labels for nodes in order to uniquely identify them within a children list. In the case of LastMerge, these identifiers are used to speed up the matching process and to increase merge accuracy. Despite the fact that which nodes can have an identifier is clearly language dependent, the extraction of these identifiers also depends on the node structure itself.

For example, in Java, a method declaration can be uniquely identified within a class by its signature — its name and parameters types. On the other hand, a field declaration can also be uniquely identified solely by its name. This means that the tool knowing which nodes of the language are uniquely identifiable is not enough; the procedure used to extract such identifier from within the node must also be provided in the configuration.

While configuring LastMerge, for each identifiable node, the user must inform how these identifiers can be extracted using *Tree Sitter* queries. These queries consist of a set of patterns that match specific nodes and their children, allowing the extraction of relevant information from the tree structure. They are executed during the parsing stage to locate specific code structures within nodes through pattern matching and assign them as the node identifier. Figure 15 illustrates an example *Tree Sitter* representation of a class declaration alongside a query that extracts the class name, which can be used to uniquely identify it within the context of a Java file.

5.2 INSTANTIATING LASTMERGE FOR OTHER LANGUAGES

In this section, we present a configuration for LastMerge for usage with C#. As it is a language that is used in industry and, to the best of our knowledge, has no language specific

structured tool implemented, the language is a good candidate for evaluating the effort of instantiating LastMerge for a new language. The presented configuration will be succinct, in the sense that we focus in covering only the most common and basic aspects of the language. Our goal is to estimate the effort of instantiating LastMerge for a new language, and better understand the challenges involved in such process. The configuration for the tool to support C# has been integrated through a pull request to the official LastMerge repository.[3]

For source code parsing, we use the official *Tree Sitter* grammar for C#.[4], which is based on the Roslyn grammar (the official .NET compiler platform). The grammar features a comprehensive set of rules for parsing C# source code, including support for many versions of C#, including the latest ones. Despites providing a complete and accurate representation of the C# language syntax, it differs from the official C# grammar in some aspects. Such changes involve adherence to *Tree Sitter* conventions, but also simplifications to the resulting tree, in order to reduce parsing states and complexity.

The next step of the configuration process involves specifying semantical aspects of the language such as node reordering and identifier extraction. For example, just like in Java, C# allows declarations within a `class` or an `interface` to be reordered without prejudice to program semantics. One of the challenges that arise during this process however is to identify the node kinds that represent such declarations in the *Tree Sitter* grammar. In principle, one might expect that deep knowledge of the *Tree Sitter* grammar is required to identify such information. However, *Tree Sitter* provides a playground[5] for users to explore the grammar interactively by providing a source code and inspecting its resulting tree. Figure 16 shows how the playground can be used to inspect the resulting tree for a given piece of code and identify segments of the tree and its corresponding node kinds. In special, we use the playground to identify that, for C#, the declarations within a `class` are grouped under a node of kind `declaration_list` — a later inspection shows that this is also the case for `interface` declarations.

We continue the process by specifying how identifiers for nodes are extracted. As previously stated, this is done by providing a dictionary that maps node kinds to *Tree Sitter* queries that are executed to extract identifiers from the nodes through pattern matching. Once more, just like in Java, C# allows method declarations to be uniquely identified by their signature, which

[3] <https://github.com/jpedroh/last-merge/pull/70>
[4] <https://github.com/tree-sitter/tree-sitter-c-sharp>
[5] <https://tree-sitter.github.io/tree-sitter/7-playground.html>

Figura 16 – A screenshot from the *Tree Sitter* playground. On top, the user can provide source code in a language. At the bottom, the user can visualize the resulting tree after parsing. Clicking on the tree nodes will highlight the corresponding code segment. This way, the user can inspect the resulting tree and identify relevant aspects about its structure.

consists of the method name and its parameters types. Field declarations within a class, on the other hand, can be uniquely identified by their name alone. Once more, the *Tree Sitter* playground is useful to identify the node kinds that represent such declarations. Figure 17 shows how it can also be used to craft and test the *Tree Sitter* queries that will be used during parsing for extracting identifiers.

Figura 17 – A screenshot from the *Tree Sitter* playground. When on query mode, the user can provide a *Tree Sitter* query that will be evaluated by the playground. On the top `amount` is highlighted in blue as it is matched by the `@name` capture from the query.

Once we have identified the node kinds whose children can be safely reordered and the nodes that can have unique labels assigned — as well as specifying how such extraction occurs, our configuration is complete. Figure 18 summarizes the resulting configuration for C#. Currently, the configuration is directly tied into the source code of LastMerge, but we plan to support external configuration files in the future.

```
let c_sharp_configuration = LanguageConfiguration {
  // The tree-sitter declarations for usage for parsing C#
  language: tree_sitter_c_sharp::LANGUAGE.into(),
  // We won't stop compilation at any node kind, treat only literals as leaf nodes
  stop_compilation_at: HashSet::new(),
  // We allow reordering of children only for these node kinds
  kinds_with_unordered_children: HashSet::new(["declaration_list", "enum_member_declaration_list"]),
  // No parsing handlers implemented for C#
  parsing_handlers: ParsingHandlers::empty(),
  // Each entry associates a node kind with a query that extracts identifiers
  identifier_extractors: IdentifierExtractors::from([
    IdentifierExtractor::of(
      "constructor_declaration",
      "(constructor_declaration name: (identifier) @method_name parameters: (parameter_list ([ (parameter type: _@parameter_type) ] "," ?) *))"
    ),
    IdentifierExtractor::of(
      "method_declaration",
      "(method_declaration name: (identifier) @method_name parameters: (parameter_list ([ (parameter type: _@parameter_type) ] "," ?) *))"
    ),
    IdentifierExtractor::of("class_declaration","(class_declaration (identifier) @class_name)"),
    IdentifierExtractor::of("enum_declaration","(enum_declaration (identifier) @class_name)"),
    IdentifierExtractor::of("interface_declaration","(interface_declaration (identifier) @class_name)"),
    IdentifierExtractor::of("variable_declaration","(variable_declarator (identifier) @name)"),
  ])
},
```

Figura 18 – Rust pseudocode of the configuration of LastMerge for usage with C#. The resulting code is more verbose, so some lines and function calls have been simplified for better visualization.

We also conduct a small sanity check to verify that the configuration works as expected. We create a small C# merge scenario that involves some of the most well known situations where structured merge thrives. Figure 19 shows the resulting merge scenario and its resolution by LastMerge. In this scenario, both branches add code to the same region of the base, but in different ways. Each parent adds both a property and a getter to the same class within the same region, but with different signatures. Furthermore, both of them also modify different tokens of an expression within the same line: the calculation performed in the `MultiplyNumbers` method. Finally, both of them also adds a new property `name` but with different initial values in each parent — which should result in a conflict. The scenario is successfully resolved by LastMerge, with spurious conflicts not being reported and actual conflicts highlighted.

The entire process of instantiating LastMerge for C# is succinct and straightforward. Whereas the pull request that introduces the configuration consists of around 90 lines of code,

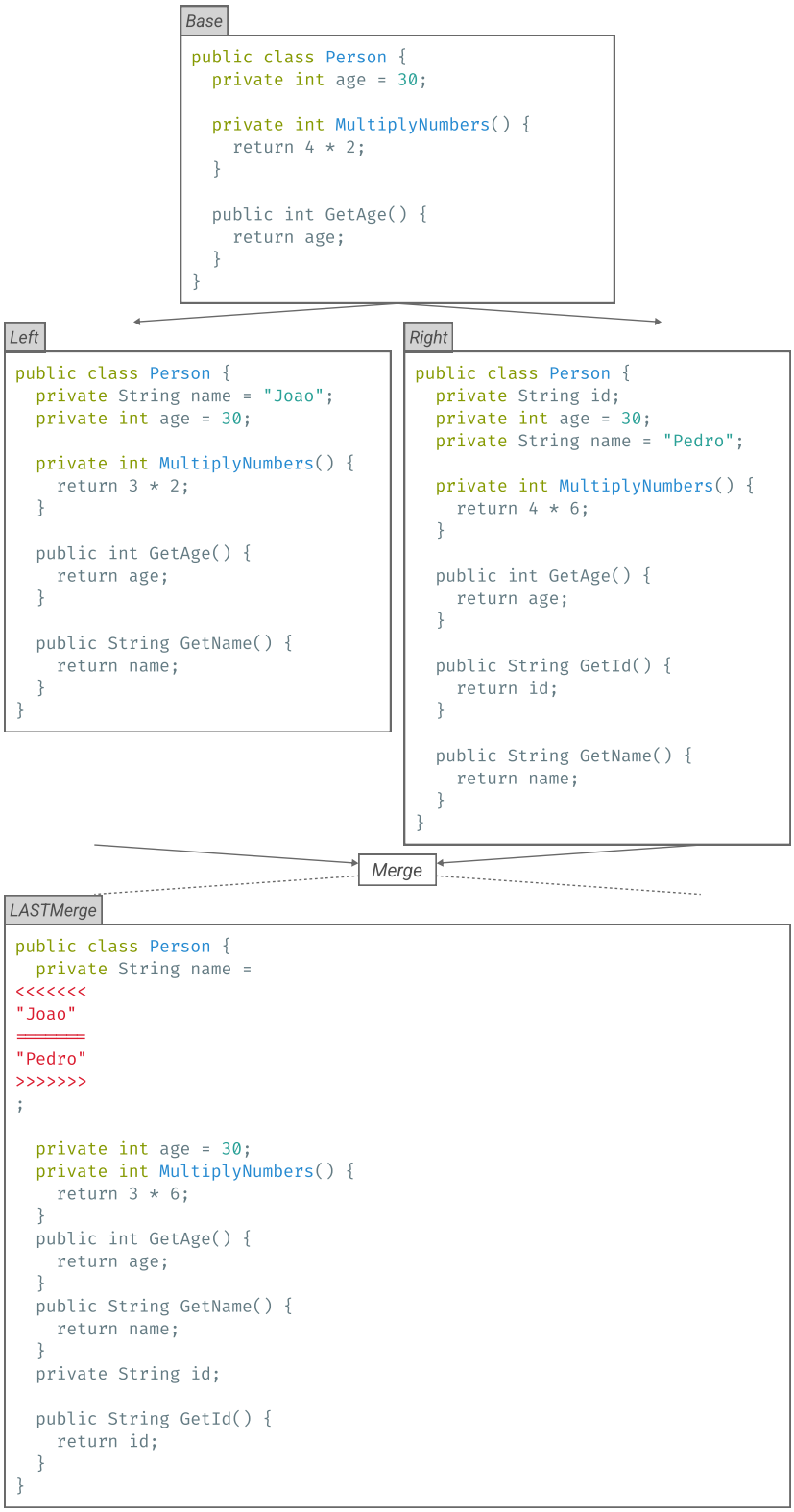

Figura 19 – A small C# merge scenario that involves some of the most well known situations where structured merge thrives. The scenario is successfully resolved by LastMerge (spurious conflicts are not reported and actual conflicts are highlighted).

instantiating any of the existing state of the art tools in the same way would be impractical. At the end of this process, we argue that instantiating LastMerge for a new language rises as a tremendously simple process. To implement a new language-specific structured merge tool, one would need to either implement the parsing, matching, and merging algorithms from scratch, or would need to dive deep in its existing implementation details in order to augment it. In the case of LastMerge, however, the configuration is based on a high level description of language specific aspects, so a user with fairly basic knowledge of the language and some high level knowledge of compilers (grammars and CSTs) would certainly be able to reproduce such minimal configuration at most within a couple of hours. Nonetheless, one of the key benefits is that the configuration can also be iterated and improved over time as LastMerge is used by the developer on its day-to-day. By being exposed to the results of the tool in a broader range of merge scenarios, the user can identify missing pieces of configuration or how the existing one can be tweaked to better suit their needs.

5.3 LIMITATIONS OF LASTMERGE FOR INSTANTIATION

Despites the simplicity of the configuration process, LastMerge still has some limitations that currently prevents it from being used with any programming language. One such limitation arises from its pretty printing mechanism. After merging the files, LastMerge converts the resulting tree into a textual representation through a pretty printing process. Due to its prototypical nature, our algorithm for pretty printing solely pipes the tree into a single chunk of text separated by spaces, as shown in Figure 20. This way, the resulting text does not preserve the original formatting of the source code, such as indentation and line breaks.

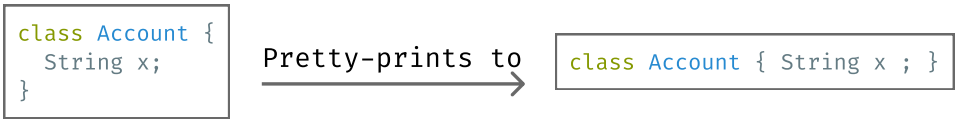

Figura 20 – On the left the original source code with a certain formatting applied. At the right, the resulting source code after the execution of LastMerge's pretty printing. Note that the pretty printer does not respect the original formatting of the source code, and simply outputs a single chunk of text separating each token by a whitespace.

This limitation is not a problem for languages that do not rely on formatting, such as C# and Java. As each statement can be perfectly delimited by using semicolons, these languages do not require specific indentation or line breaks to be valid. That is not the case for languages such

as Python, which relies on indentation to define code blocks. Failing to preserve indentation in such languages results in syntactically invalid code, which would not compile or run. Other limitation also arises from languages that uses mechanisms to infer the end of statements, such as JavaScript.

Javascript uses a mechanism called *automatic semicolon insertion (ASI)* to infer the end of statements. *ASI* allows developers to omit semicolons at the end of statements, and the JavaScript engine will automatically insert them where it deems necessary. This feature is designed to make the language more flexible and forgiving, allowing developers to write code without worrying too much about semicolon placement. However, it is easy to notice that LastMerge's pretty printing can result in code that is incorrect due to the way *ASI* works. To illustrate this, consider the code snippets in Figure 21. The left snippet shows the original source code, while the right ones shows the result of formatting it with LastMerge's pretty printing algorithm. On the left snippet, the developer choose to omit a semicolon after the end of the declaration of variable a and relies on *ASI* to infer it through the line break. However, as on the right snippet the pretty printer outputs a single line of code, the *ASI* mechanism will not be able to infer the end of the first statement. Thus, it interprets the entire code as a single statement, and results in a syntax error.

Figura 21 – On the left, an example snippet consisting of the declaration of two variables. Due to the line break between them, the *ASI* mechanism of JavaScript correctly interprets them as being two different statements. The snippet on the right is the result of formatting the original source code with LastMerge's algorithm. As it combines the two statements into a single line, the *ASI* mechanism will not be able to infer the end of the first statement, resulting in a syntax error.

Mitigating these issues, involves implementing a more sophisticated pretty printing algorithm that is able to respect the formatting rules of each language. State of the art structured tools, for example, do this by using a language specific pretty printer that is able to convert the resulting tree into a valid textual representation. The *trade-off* here is that these pretty-printers sometimes do not preserve the original formatting of the source code, but rather apply a set of rules to format the code according to a predefined style. Another possibility, explored by *Mergiraf*, is to use heuristics to infer the original formatting of the source code from the resulting tree. During pretty-printing, the tool uses such heuristics to calculate whitespaces

between tokens, as well as indentation levels, based on the original revisions of the source code. Despites being more complex, this approach is able to yield syntactically valid code in a language independent way, and arises as a promising direction for future improvement of LastMerge.

5.4 SUMMARY

In this chapter, we presented the configuration options available in LastMerge for adapting the tool to different programming languages. We discussed the interface for configuring language-specific settings, the process of instantiating the tool for a new language, and the known limitations that users should be aware of when using LastMerge with different programming languages.

We demonstrated that instantiating LastMerge for a new language, such as C#, is a low-effort process that can be accomplished with minimal knowledge of the tool's internals. One can argue that the effort involved in instantiating LastMerge should be similar for languages that share the syntactical and semantical characteristics of Java, such as C#, Kotlin and C++. Nonetheless, we conjecture that the effort required to instantiate it for languages with different syntactical and semantical characteristics, such as Python or JavaScript, will be higher, but still manageable.

In fact, one of the main limitations for the usage of LastMerge is not the effort required to instantiate it for a new language, but rather the limitations of its current implementation. As discussed in Section 5.3, languages that rely on formatting or indentation to define valid grammar may not be fully supported by LastMerge. This is the case for languages such as Python, where the tool will often yield an invalid program due to its limited pretty-printing capabilities. This highlights a path for future work, where improving the pretty-printing capabilities of LastMerge could significantly enhance its usability for languages that rely on formatting or indentation.

6 CONCLUSION

In this work, our goal was to reduce the barriers that hinder the adoption of structured merge in industry, where support for multiple languages is often needed for most nontrivial projects. To reduce these barriers, we introduced LastMerge, a *generic* structured merge tool that can be easily configured for each language. LastMerge relies on a core merge engine that operates over generic trees. Combined with a high level description of language specific aspects, developers can easily adapt LastMerge for new languages, or refine support for existing ones.

To understand wether introducing language independency impacts the accuracy and performance of structured merge, we conducted an empirical study comparing *generic* structured merge tools and state-of-the-art language-specific ones. Our comparative study involved four structured merge tools: *jDime* (APEL; LEßENICH; LENGAUER, 2012) and *Spork* (LARSEN et al., 2023), two state-of-the-art Java specific tools; and both LastMerge and *Mergiraf*, which are two *generic* structured merge tools that operate on language independent trees. The tools are compared in pairs: LastMerge with *jDime*, and *Mergiraf* with *Spork*. This pairing reflects the influence of existing state of the art, language specific, tools on the design and implementation of the generic ones. It also helps to isolate the generality aspect, mitigating bias that could arise from effects not related to implementing the generic requirement.

Our results show no evidence that introducing language independency to structured merge significantly impacts its merge accuracy or runtime performance. In fact, despites observing a difference rate of approximately 10% between the Java specific tools and their *generic* counterparts instantiated for Java, we observe that most of the differences stem from implementation details and configuration choices. Furthermore, *generic* structured merge tools exhibit comparable runtime performance to the state of the art language-specific implementations.

We also conducted a case study to understand the effort of configuring LastMerge for new languages. We instantiated a minimal configuration for C# and found that the effort is significantly lower than the one involved in implementing a language-specific tool. In fact, a user with fairly basic knowledge of the language and some high level knowledge of compilers could derive such minimal configuration within at most a few hours, compared to weeks or months for implementing a new language-specific tool. We observe however, that LastMerge still requires more improvements to support languages where code formatting is important, such as

Python or JavaScript. Nonetheless, our overall results suggests that *generic* structured merge tools can effectively replace language-specific ones, achieving similar levels of accuracy and efficiency, thus paving the way for broader adoption of structured merge in industry.

6.1 CONTRIBUTIONS

We summarize the contributions of this work as follows:

- LastMerge, a *generic* structured merge tool designed to be easily configured for different programming languages. It leverages the *Tree Sitter* parser framework to parse source code into generic trees, and provides a thin configuration interface that allows users to adapt the tool for new languages or refine support for existing ones.
- An empirical study that compares *generic* structured merge tools with *Java specific* ones. The study uses an extensive state-of-the-art dataset to evaluate the impact of language independency on the precision and performance of structured merge tools. We find no evidence that generic structured merge tools perform worse than the *Java specific* ones.

6.2 RELATED WORK

Software integration is a well studied problem, and this chapter discusses the many approaches that have been previously proposed to address it. Section 6.2.1 discusses other structured tools and how they differ from ourselves. Section 6.2.2 discusses semi-structured tools, a class of tools that aim to balance precision and performance. Finally, Section 6.2.3 discusses complementary approaches that have also been proposed to address different problems that arise during software integration.

6.2.1 Structured Merge

Several structured merge tools have been proposed previously. These tools rely on the syntax and semantics of specific programming languages to perform the merge, in contrast to LastMerge, which is designed to be more flexible and not limited to any particular language.

Apel, Leßenich and Lengauer (2012) introduced *jDime*, a tool that applies a structured merge strategy to integrate Java programs. *jDime* employs auto-tuning to dynamically switch

between structured and unstructured merge based on the presence of conflicts. Later, Leßenich et al. (2017) enhanced *jDime* by incorporating a look-ahead mechanism into the node matching algorithm. This improvement enables the tool to match nodes across different hierarchical levels, allowing it to better detect refactorings such as renamings and code movements. As a result, matching precision increased by 28% without compromising performance.

Zhu, He and Yu (2019) proposed AutoMerge. Built on top of *jDime*, it introduces a new matching heuristic that aims to quantify the quality of a matching between the nodes. This heuristic quantifies the similarity between nodes, and the algorithm seeks to maximize this quality function to avoid incorrect or unrelated matchings, thereby improving precision. Their evaluation shows that AutoMerge more closely reproduces the developer resolved merges found in commit histories compared to *jDime*, at the expense of a performance overhead.

Larsen et al. (2023) proposed *Spork*, a structured merge tool for Java. *Spork* relies on GumTree (FALLERI et al., 2014) to compute node matchings and introduces a *high-fidelity pretty-printing* mechanism that reuses code fragments from the original revisions. This enables the preservation of source code formatting, and their evaluation shows that *Spork* retained the original source code in over 90% of merged files when compared to *jDime*.

Despite their benefits, extending these existing tools to support other programming languages is challenging, primarily due to their strong coupling with the specific languages for which they were originally designed. In contrast, LastMerge operates on generic tree structures and leverages the *Tree Sitter* infrastructure to parse source code into these trees. Additionally, LastMerge provides an interface that allows users to extend the tool for use with other programming languages.

More recently, *Mergiraf*[1], another *generic* structured merge tool, has also been proposed. It is based on the algorithms and design of *Spork*, which have been adapted to work with generic trees parsed with *Tree Sitter* — just like LastMerge. It also provides a configuration interface to support new languages that is similar to the one used in LastMerge, and has already been instantiated for over 20 languages so far[2]. Our work is the first to compare the Java instantiation of *Mergiraf* with its state of the art counterpart. In our evaluation, we found that just like LastMerge and *Mergiraf* present consistent results when compared with state of the art tools. This further reinforces the idea that the *generic* structured merge tools can be as effective as language-specific ones, and that the generality aspect is not influenced by

[1] <https://mergiraf.org/>
[2] <https://mergiraf.org/languages.html>

the other design decisions of the tools.

6.2.2 Semistructured Merge

Other techniques have been explored as alternatives to structured merge. One such approach is semistructured merge, a hybrid strategy designed to balance precision and accuracy. Semistructured merge applies structured merge to higher-level elements, such as method declarations, while utilizing unstructured merge for lower-level elements, like expressions and statements within method bodies.

Apel et al. (2011) proposed FSTMerge, a generic semistructured merge engine. They have configured it for use with C#, Java, and Python. To support a new language, developers must provide an annotated grammar that specifies the elements of the language where the order does not matter. However, this approach requires a deep understanding of the language's grammar and the ability to annotate it correctly, which can pose a barrier for many users. In contrast, LastMerge relies on the *Tree Sitter* parser framework, which has been instantiated for over 350 languages, and offers a thin configuration interface for extending the tool for usage with new languages. This makes LastMerge more flexible and easier to adapt to new languages compared to FSTMerge.

Cavalcanti, Borba and Accioly (2017) evaluated FSTMerge by comparing it with unstructured merge on Java code. They found that FSTMerge significantly reduces the number of false positives, but this comes at the expense of missing actual conflicts (false negatives). Based on their findings, they proposed s3m, a semistructured merge tool for Java built on top of FSTMerge. s3m extends FSTMerge by introducing *handlers* that update the final tree using heuristics based on information gathered during the merge process. These *handlers* enhance merge accuracy by addressing cases where FSTMerge previously produced additional false positives and false negatives, such as in renaming scenarios. We draw inspiration from s3m to implement the *parsing handlers* in LastMerge.

Tavares et al. (2019) explored the usage of semistructured merge in the context of JavaScript, by extending FSTMerge to support it and build jsFSTMerge. During this process, some limitations and difficulties of extending FSTMerge were identified. The first one is that, differently from LastMerge, FSTMerge needs the user to manually annotate the grammar of the language, which can be a complex and error-prone task. Also, the tool is unable to handle the combination of ordered and unordered nodes at the same level of the

tree, which is something common in scripting languages like JavaScript. They conduct an empirical evaluation comparing semistructured merge with unstructured merge on 10,345 merge scenarios from 50 JavaScript projects on GitHub. Their results shows that the benefits of using semistructured merge in JavaScript are far less pronounced than in Java, with their best instantiation of FSTMerge achieving a mere 6% reduction in the number of conflicts compared to unstructured merge. Such findings suggest that the benefits of semistructured merge may not be as significant in scripting languages as they are in more verbose and structured languages like Java.

More recently, Cavalcanti et al. (2024) proposed Sesame, a tool designed to emulate structured merge by leveraging semistructured merge along with language-specific syntactic separators. The tool uses semistructured merge to integrate revisions; however, before performing unstructured merge on lower-level elements, it synthetically splits the source code based on these language separators, such as curly braces in Java. This splitting infers a structure that, when combined with unstructured merge, emulates structured merge. Their empirical evaluation demonstrates that Sesame achieves results comparable to those of structured tools, positively by reducing false positives and negatively by increasing the number of false negatives.

Despite being a structured merge tool, LastMerge can emulate the semistructured behavior of both FSTMerge, jsFSTMerge, s3m and Sesame. This is possible because the user can configure the tool to stop parsing at an intermediate tree level, treating all descendants of that node as a single textual element.

6.2.3 Complementary Approaches

Despites the best of their efforts, unstructured, structured and semistructured merge tools might still miss actual conflicts. Such conflicts are often referred to as *semantic conflicts*, and arise when the changes introduced by developers are not syntactically conflicting but still interfere with each other in a way that is not immediately apparent from the code structure alone. Detecting these conflicts requires a deeper understanding of the code's semantics, and different approaches have been proposed to address this problem. They can be separated into two main categories: those that rely on static analysis and those that rely on dynamic analysis.

Jesus et al. (2024) explores the usage of *statical analysis* to detect semantic conflicts. Although this approach has already been explored in the past, it is often impractical due to

the prohibitive cost of such analysis. They propose a lightweight static analysis tool that aims to detect semantic conflicts in Java code. It does so by running static analysis on the merged code to identify potential violations in data flow, assignments and dependencies. They evaluate their tool on a set of 99 experimental units, associated with 54 merge scenarios extracted from 39 projects. Their results show that the tool is able to achieve an accuracy of 60% on the scenarios, and a F1 Score of 50%.

Silva et al. (2020) follows a different approach by using *dynamic analysis* to detect semantic conflicts. In their work, they propose a tool that aims to detect semantic conflicts by trying to automatically detect changes in the behavior of the merged code. The tool does so by heuristically building partial specifications of the different versions through the automatic generation of test suites. In their work, they use widely recognized automated tools such as Evosuite and Randoop for such generation and later execute these tests on the different versions of the merge scenario. Depending on the results of these tests in each version, the tool is able to identify behavior changes and potentially identify semantic conflicts. They conduct an empirical evaluation of their technique on 40 different scenarios. A manual analysis of the results shows that their tool is able to not report any false positives on the sample. However, it fails to detect 11 of the actual 15 existing semantic conflicts, resulting in a low recall of only 26%.

Both the tools proposed by Jesus et al. (2024) and Silva et al. (2020) can be used in conjunction with LastMerge. In this case, LastMerge would be used to perform the initial merge, and then these tools would be invoked to analyze the merged code for potential semantic conflicts. This approach allows developers to leverage the strengths of both structured merge and semantic conflict detection, providing a more comprehensive solution for handling complex merge scenarios.

6.3 FUTURE WORK

In this work, we have focused on the design and implementation of LastMerge, a generic structured merge tool. Furthermore, we have conducted an empirical study comparing *generic* merge tools with *language-specific* ones. Despites the promising results, we acknowledge that there are still many opportunities for future work to enhance LastMerge and better understand the applicability of structured merge across different programming languages. We identify the following avenues for future work:

- One of the main limitations of LastMerge for production usage is its rudimentary pretty-printing capabilities. Due to the way its algorithm works, the tool can yield invalid programs for languages where formatting is significant, such as Python. Future work should focus on improving the pretty-printing mechanism to ensure that the merged code better retains the syntactical and semantical rules of the original source code. This would enhance the usability of LastMerge across more programming languages.

- The accuracy of structured merge is highly dependent on the quality of its *matching* process. Matching nodes incorrectly can lead to false positives and false negatives, which can significantly impact the merge results. Future work could explore the implementation of different algorithms and heuristics to improve the quality of the matching stage in LastMerge.

- Despites structured merge being a promising approach, it is still unclear wether its benefits generalize to other programming languages. Future work can focus on instantiating LastMerge for other programming languages, and evaluating its performance and precision in these languages compared to other strategies. This would help to understand the limitations and strengths of structured merge across different programming languages.

BIBLIOGRAPHY

APEL, S.; LEßENICH, O.; LENGAUER, C. Structured merge with auto-tuning: balancing precision and performance. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2012. p. 120–129.

APEL, S.; LIEBIG, J.; BRANDL, B.; LENGAUER, C.; KäSTNER, C. 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*. New York, NY, USA: Association for Computing Machinery, 2011. (ESEC/FSE '11), p. 190–200. ISBN 9781450304436. Available at: <https://doi.org/10.1145/2025113.2025141>.

BRUNSFELD, M. *tree-sitter/tree-sitter: v0.25.3*. Zenodo, 2025. Available at: <https://doi.org/10.5281/zenodo.14969376>.

BUFFENBARGER, J. Syntactic software merging. In: ESTUBLIER, J. (Ed.). *Software Configuration Management*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995. p. 153–172. ISBN 978-3-540-47768-6.

CAVALCANTI, G.; BORBA, P.; ACCIOLY, P. Evaluating and improving semistructured merge. *Proc. ACM Program. Lang.*, Association for Computing Machinery, New York, NY, USA, v. 1, n. OOPSLA, Oct. 2017. Available at: <https://doi.org/10.1145/3133883>.

CAVALCANTI, G.; BORBA, P.; ANJOS, L. d.; CLEMENTINO, J. Semistructured merge with language-specific syntactic separators. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2024. (ASE '24), p. 1032–1043. ISBN 9798400712487. Available at: <https://doi.org/10.1145/3691620.3695483>.

CAVALCANTI, G. J. d. C. *Should we replace our merge tools?* Phd Thesis (PhD Thesis) — Universidade Federal de Pernambuco, 2019.

CLEMENTINO, J.; BORBA, P.; CAVALCANTI, G. Textual merge based on language-specific syntactic separators. In: *Proceedings of the XXXV Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021. (SBES '21), p. 243–252. ISBN 9781450390613. Available at: <https://doi.org/10.1145/3474624.3474646>.

CONRADI, R.; WESTFECHTEL, B. Version models for software configuration management. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 30, n. 2, p. 232–282, Jun. 1998. ISSN 0360-0300. Available at: <https://doi.org/10.1145/280277.280280>.

CVS. 2025. Available at: <https://www.nongnu.org/cvs/>.

FALLERI, J.-R.; MORANDAT, F.; BLANC, X.; MARTINEZ, M.; MONPERRUS, M. Fine-grained and accurate source code differencing. In: *Proceedings of the International Conference on Automated Software Engineering*. [s.n.], 2014. p. 313–324. Available at: <https://hal.archives-ouvertes.fr/hal-01054552/file/main.pdf>.

GIT. 2025. Available at: <https://git-scm.com/>.

GITHUB. *Greatest Hits*. 2020. <https://archiveprogram.github.com/greatest-hits/>.

GITHUB. *The State of Open Source in 2024*. 2024. Available at: <https://github.blog/news-insights/octoverse/octoverse-2024/>.

HUNT, J.; TICHY, W. Extensible language-aware merging. In: *International Conference on Software Maintenance, 2002. Proceedings.* [S.l.: s.n.], 2002. p. 511–520.

JESUS, G. S. D.; BORBA, P.; BONIFáCIO, R.; OLIVEIRA, M. B. D. Lightweight semantic conflict detection with static analysis. In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. New York, NY, USA: Association for Computing Machinery, 2024. (ICSE-Companion '24), p. 343–345. ISBN 9798400705021. Available at: <https://doi.org/10.1145/3639478.3643118>.

KHANNA, S.; KUNAL, K.; PIERCE, B. C. A formal investigation of diff3. In: ARVIND, V.; PRASAD, S. (Ed.). *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 485–496. ISBN 978-3-540-77050-3.

KUHN, H. W. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, v. 2, n. 1-2, p. 83–97, 1955. Available at: <https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109>.

LARSEN, S.; FALLERI, J.-R.; BAUDRY, B.; MONPERRUS, M. Spork: Structured Merge for Java With Formatting Preservation . *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 49, n. 01, p. 64–83, Jan. 2023. ISSN 1939-3520. Available at: <https://doi.ieeecomputersociety.org/10.1109/TSE.2022.3143766>.

LEßENICH, O.; APEL, S.; KäSTNER, C.; SEIBT, G.; SIEGMUND, J. Renaming and shifted code in structured merging: Looking ahead for precision and performance. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2017. p. 543–553.

MCNEMAR, Q. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, Springer Science and Business Media LLC, v. 12, n. 2, p. 153–157, Jun. 1947.

MENS, T. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, v. 28, n. 5, p. 449–462, 2002.

MERCURIAL. 2025. Available at: <https://www.mercurial-scm.org/>.

MUNAIAH, N.; KROH, S.; CABREY, C.; NAGAPPAN, M. Curating github for engineered software projects. *Empirical Softw. Engg.*, Kluwer Academic Publishers, USA, v. 22, n. 6, p. 3219–3253, Dec. 2017. ISSN 1382-3256. Available at: <https://doi.org/10.1007/s10664-017-9512-6>.

PEREIRA, R.; COUTO, M.; RIBEIRO, F.; RUA, R.; CUNHA, J.; FERNANDES, J. a. P.; SARAIVA, J. a. Energy efficiency across programming languages: how do energy, time, and memory relate? In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (SLE 2017), p. 256–267. ISBN 9781450355254. Available at: <https://doi.org/10.1145/3136014.3136031>.

RIGBY, P. C.; BARR, E. T.; BIRD, C.; GERMAN, D. M.; DEVANBU, P. Collaboration and governance with distributed version control. *ACM Transactions on Software Engineering and Methodology*, v. 5, 2009.

SCHESCH, B.; FEATHERMAN, R.; YANG, K. J.; ROBERTS, B.; ERNST, M. D. Evaluation of version control merge tools. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2024. (ASE '24), p. 831–83. ISBN 9798400712487. Available at: <https://doi.org/10.1145/3691620.3695075>.

SEIBT, G.; HECK, F.; CAVALCANTI, G.; BORBA, P.; APEL, S. Leveraging structure in software merge: An empirical study. *IEEE Transactions on Software Engineering*, v. 48, n. 11, p. 4590–4610, 2022.

SILVA, L.; BORBA, P.; MAHMOOD, W.; BERGER, T.; MOISAKIS, J. Detecting semantic conflicts via automated behavior change detection. In: *36th IEEE International Conference on Software Maintenance and Evolution (ICSME 2020)*. [S.l.: s.n.], 2020. p. 174–184.

SILVA, L. D.; BORBA, P.; PIRES, A. Build conflicts in the wild. *J. Softw. Evol. Process*, John Wiley & Sons, Inc., USA, v. 34, n. 4, Apr. 2022. ISSN 2047-7473. Available at: <https://doi.org/10.1002/smr.2441>.

SUBVERSION. 2025. Available at: <https://subversion.apache.org/>.

TAVARES, A. T.; BORBA, P.; CAVALCANTI, G.; SOARES, S. Semistructured merge in javascript systems. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2019. p. 1014–1025.

WESTFECHTEL, B. Structure-oriented merging of revisions of software documents. In: *Proceedings of the 3rd International Workshop on Software Configuration Management*. New York, NY, USA: Association for Computing Machinery, 1991. (SCM '91), p. 68–79. ISBN 0897914295. Available at: <https://doi.org/10.1145/111062.111071>.

YANG, W. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., USA, v. 21, n. 7, p. 739–755, Jun. 1991. ISSN 0038-0644. Available at: <https://doi.org/10.1002/spe.4380210706>.

ZHU, F.; HE, F. Conflict resolution for structured merge via version space algebra. *Proc. ACM Program. Lang.*, Association for Computing Machinery, New York, NY, USA, v. 2, n. OOPSLA, Oct. 2018. Available at: <https://doi.org/10.1145/3276536>.

ZHU, F.; HE, F.; YU, Q. Enhancing precision of structured merge by proper tree matching. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. [S.l.: s.n.], 2019. p. 286–287.