Merge and Code Review

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

Textual merge, unstructured merge

$A = [1, 4, 5, 2, 3, 6]$
$O = [1, 2, 3, 4, 5, 6]$
$B = [1, 2, 4, 5, 3, 6]$

(a) inputs

A	1	4	5	2	3			6
O	1			2	3	4	5	6

O	1	2	3	4	5		6
B	1	2		4	5	3	6

(b) maximum matches

A	1	4,5	2	3	6
O	1		2	3,4,5	6
B	1		2	4,5,3	6

(c) `diff3` parse

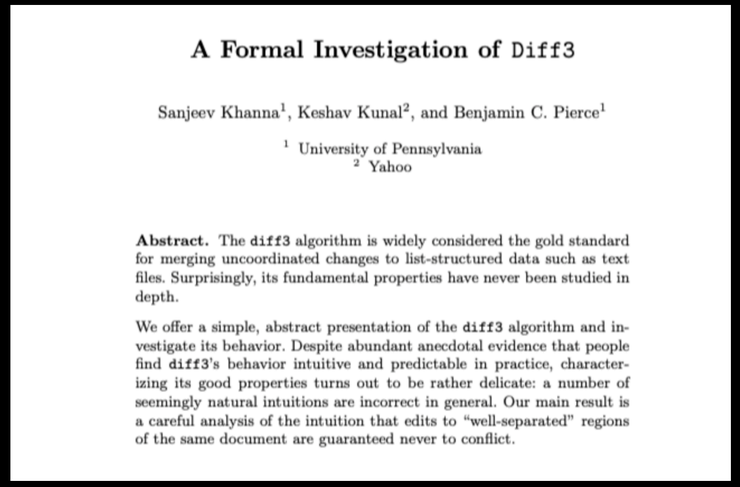
A'	1	4,5	2	3	6
O'	1	4,5	2	3,4,5	6
B'	1	4,5	2	4,5,3	6

(d) calculated output

```
1
4
5
2
<<<<<<< A
3
||||||| O
3
4
5
=======
4
5
3
>>>>>>> B
6
```

(e) printed output

https://dl.acm.org/doi/10.5555/1781794.1781836



A Formal Investigation of `Diff3`

Sanjeev Khanna[1], Keshav Kunal[2], and Benjamin C. Pierce[1]

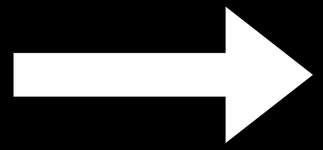
[1] University of Pennsylvania
[2] Yahoo

Abstract. The `diff3` algorithm is widely considered the gold standard for merging uncoordinated changes to list-structured data such as text files. Surprisingly, its fundamental properties have never been studied in depth.

We offer a simple, abstract presentation of the `diff3` algorithm and investigate its behavior. Despite abundant anecdotal evidence that people find `diff3`'s behavior intuitive and predictable in practice, characterizing its good properties turns out to be rather delicate: a number of seemingly natural intuitions are incorrect in general. Our main result is a careful analysis of the intuition that edits to "well-separated" regions of the same document are guaranteed never to conflict.

There is no unambiguous notion of same line when comparing files!

1
2
3
4

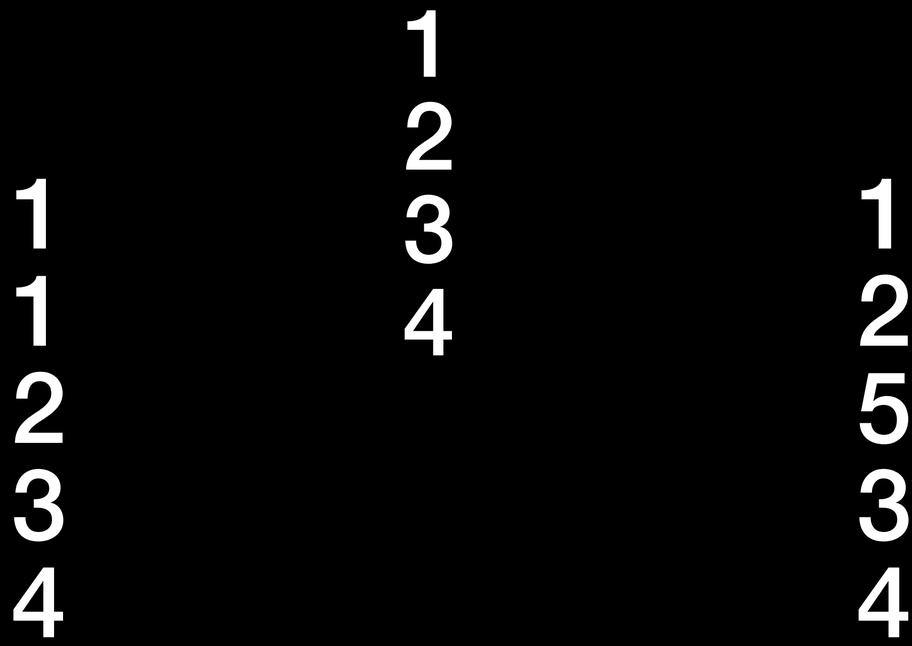


1
1
2
3
4

Added 1 before or after original 1?

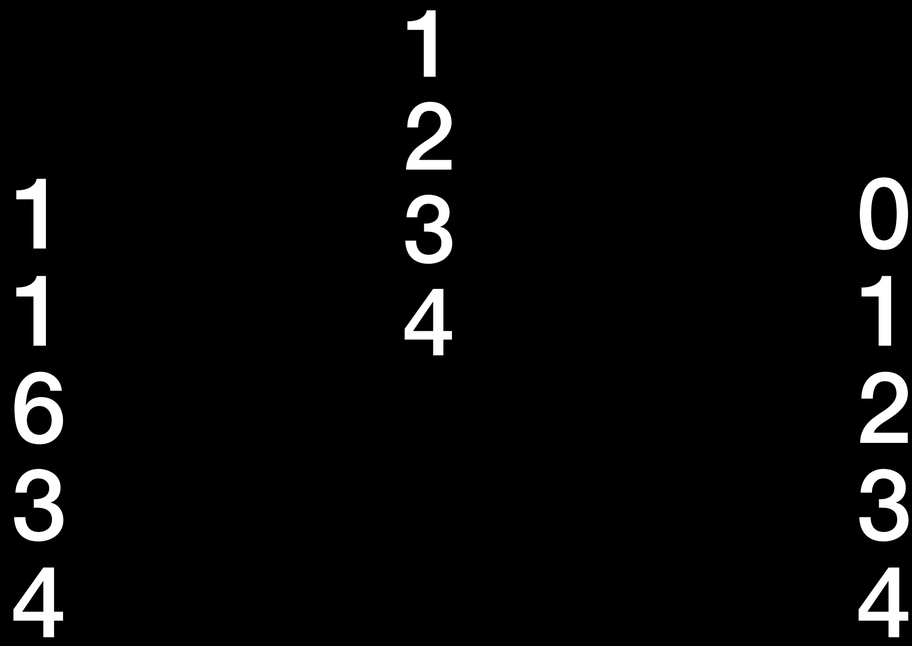
Dealing with nodes instead of lines could solve the problem.

Merge?



Longest common sequence, not always unique

Merge?



Different line-based textual merge algorithms can yield different matchings and results!

Could we have textual merge tools that are not line-based?

Could we use other units of matching besides lines?

Could we use other separators besides \n?

CSDiff, language-specific syntactic separators: { } ; ( ) ,

https://pauloborba.cin.ufpe.br/publication/2021textual_merge_based_on_language-specific_syntactic_separators/

Textual merge based on language-specific syntactic separators

Jônatas Clementino
Centro de Informática
Universidade Federal de Pernambuco
Brasil
joc@cin.ufpe.br

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco
Brasil
phmb@cin.ufpe.br

Guilherme Cavalcanti
Instituto Federal de Alagoas
Brasil
guilherme.cavalcanti@ifal.edu.br

ABSTRACT

In practice, developers mostly use purely textual, line-based, merge tools. Such tools, however, often report false conflicts. Researchers have then proposed AST-based tools that explore language syntactic structure to reduce false conflicts. Nevertheless, these approaches might negatively impact merge performance, and demand the creation of a new tool for each language. Trying to simulate the behavior of AST-based tools without their drawbacks, this paper proposes and analyzes a purely textual, separator-based, merge tool that aims to simulate AST-based tools by considering programming language syntactic separators, instead of just lines, when comparing and merging changes. The obtained results show that the separator-based textual approach might reduce the number of false conflicts when compared to the line-based approach. The new solution makes room for future studies and hybrid merge tools.

ACM Reference Format:
Jônatas Clementino, Paulo Borba, and Guilherme Cavalcanti. 2021. Textual merge based on language-specific syntactic separators. In *Brazilian Symposium on Software Engineering (SBES '21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3474624.3474646

Apesar desses problemas, a abordagem de *merge* mais utilizada na indústria atualmente ainda é o *merge* não estruturado [16, 19], que se utiliza de uma análise puramente textual, equiparando linha a linha arquivos com código a ser integrado, e assim detectar conflitos ou realizar a integração. Porém, muitas vezes essa abordagem sinaliza falsos conflitos, fazendo com que desenvolvedores percam tempo ao resolvê-los. Por causa disso, pesquisadores propuseram ferramentas que exploram a estrutura sintática do código que está sendo integrado, criando árvores sintáticas a partir do texto dos arquivos [1, 2], e assim obtendo melhor acurácia no *merge*. Essas abordagens são chamadas estruturadas e semiestruturadas, no caso em que as árvores são parciais e alguns elementos sintáticos, como declaração de método, são representados como texto. Estudos anteriores [2, 9, 11, 22] comparam essas duas abordagens (estruturada e semiestruturada) em relação à não estruturada e mostram que, para a maioria dos projetos e situações de *merge*, há uma redução de conflitos em favor da semiestruturada e da estruturada. Essa redução se dá por conta de falsos conflitos que elas resolvem automaticamente, como, por exemplo, quando os desenvolvedores adicionam dois métodos diferentes e independentes numa mesma área de texto no código [2, 10].

Reduces false positives

```
1  public String toString(List<T> l) {
2      if(l.size() == 0) { return ""; }
3      return String.join(",", l);
4  }
```

Figure 1: Arquivo *base* que contém o método toString

```
1  public String toString(List<T> l) {
2      if(l == null || l.isEmpty()) { return ""; }
3      return String.join(",", l);
4  }
```

Figure 2: Arquivo *left* que contém o método toString

```
1  public String toString(List<T> l) {
2      if(l.size() == 0) { return D; }
3      return String.join(",", l);
4  }
```

Figure 3: Arquivo *right* que contém o método toString

```
1  public String toString(List<T> l) {
2  <<<<<<< left.java
3      if (l == null || l.isEmpty()) { return ""; }
4  ||||||| base.java
5      if (l.size() == 0) { return ""; }
6  =======
7      if (l.size() == 0) { return D; }
8  >>>>>>> right.java
9      return String.join(",", l);
10 }
```

diff3 on preprocessed files, post processing result

```
1  public String toString(List<T> l)
2  $$$$$$${
3  $$$$$$$
4      if(l.size() == 0)
5  $$$$$$${
6  $$$$$$$ return "";
7  $$$$$$$}
8  $$$$$$$
9      return String.join(",", l);
10
11 $$$$$$$}
12 $$$$$$$
```

```
1  public String toString(List<T> l)
2  $$$$$$${
3  $$$$$$$
4      if(l == null || l.isEmpty())
5  $$$$$$${
6  $$$$$$$ return D;
7  $$$$$$$}
8  $$$$$$$
9      return String.join(",", l);
10
11 $$$$$$$}
12 $$$$$$$
```

Figure 9: Arquivo resultante de executar o Diff3 nos arquivos temporários

```
1  public String toString(List<T> l) {
2      if (l == null || l.isEmpty()) { return D; }
3      return String.join(",", l);
4  }
```

Figure 10: Resultado final do CSDiff

Reduces FPs, increases FNs

Table 1: Resultados do CSDiff em comparação ao Diff3, onde CSDiff+ representa o CSDiff com os seis separadores '{', '}', ',', '(', ')', e ';'; e CSDiff- com os três primeiros separadores ('{', '}', ';').

#	Diff3	CSDiff+	CSDiff-
Conflitos	74096	152490	99874
Arquivos com Conflitos	8670	8598	8654
Arquivos com Conflitos (sem cenários *outliers*)	1491	1419	1475
Cenários com Conflitos	595	564	583

Table 2: Resultados do CSDiff+ em comparação ao Diff3

#	Diff3	CSDiff+
Cenários com conflitos exclusivos	34	3
Arquivos com conflitos exclusivos	78	6
aFP Cenários	30	3
aFP Arquivos	63	6
aFN Cenários	0	4
aFN Arquivos	0	15

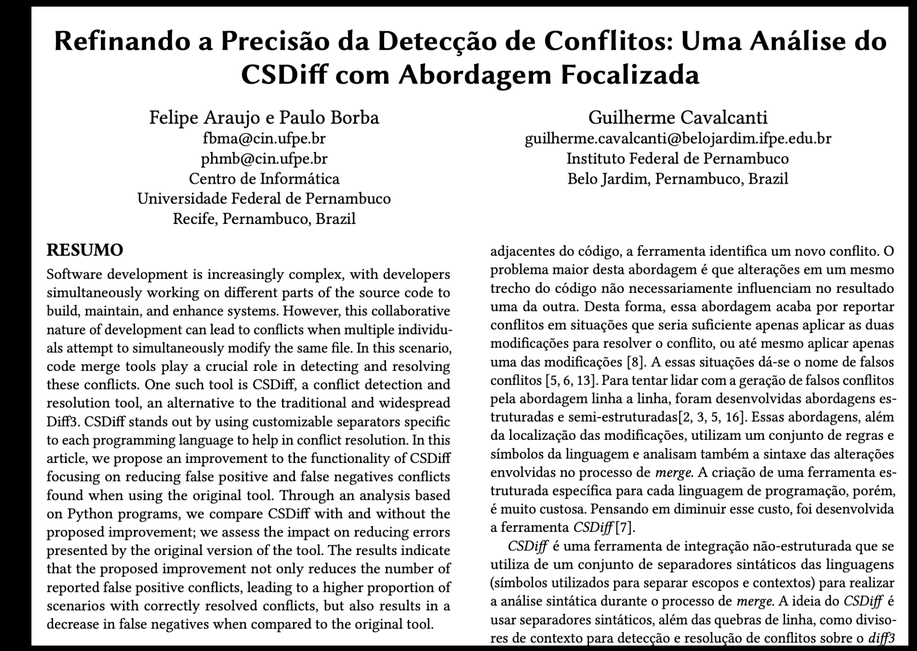
Miss alignment is the main problem

SepMerge: Avoiding markers, considering indentation, and autotuning (runs on the conflict text)

```
def to_string
(
l
:
 List[str]
)
 -> str
:

    if len
(
l
)
 == 0
:

        return ""
    return "..".join
(
l
)
```



Refinando a Precisão da Detecção de Conflitos: Uma Análise do CSDiff com Abordagem Focalizada

Felipe Araujo e Paulo Borba
fbma@cin.ufpe.br
phmb@cin.ufpe.br
Centro de Informática
Universidade Federal de Pernambuco
Recife, Pernambuco, Brazil

Guilherme Cavalcanti
guilherme.cavalcanti@belojardim.ifpe.edu.br
Instituto Federal de Pernambuco
Belo Jardim, Pernambuco, Brazil

RESUMO

Software development is increasingly complex, with developers simultaneously working on different parts of the source code to build, maintain, and enhance systems. However, this collaborative nature of development can lead to conflicts when multiple individuals attempt to simultaneously modify the same file. In this scenario, code merge tools play a crucial role in detecting and resolving these conflicts. One such tool is CSDiff, a conflict detection and resolution tool, an alternative to the traditional and widespread Diff3. CSDiff stands out by using customizable separators specific to each programming language to help in conflict resolution. In this article, we propose an improvement to the functionality of CSDiff focusing on reducing false positive and false negatives conflicts found when using the original tool. Through an analysis based on Python programs, we compare CSDiff with and without the proposed improvement; we assess the impact on reducing errors presented by the original version of the tool. The results indicate that the proposed improvement not only reduces the number of reported false positive conflicts, leading to a higher proportion of scenarios with correctly resolved conflicts, but also results in a decrease in false negatives when compared to the original tool.

adjacentes do código, a ferramenta identifica um novo conflito. O problema maior desta abordagem é que alterações em um mesmo trecho do código não necessariamente influenciam no resultado uma da outra. Desta forma, essa abordagem acaba por reportar conflitos em situações que seria suficiente apenas aplicar as duas modificações para resolver o conflito, ou até mesmo aplicar apenas uma das modificações [8]. A essas situações dá-se o nome de falsos conflitos [5, 6, 13]. Para tentar lidar com a geração de falsos conflitos pela abordagem linha a linha, foram desenvolvidas abordagens estruturadas e semi-estruturadas[2, 3, 5, 16]. Essas abordagens, além da localização das modificações, utilizam um conjunto de regras e símbolos da linguagem e analisam também a sintaxe das alterações envolvidas no processo de *merge*. A criação de uma ferramenta estruturada específica para cada linguagem de programação, porém, é muito custosa. Pensando em diminuir esse custo, foi desenvolvida a ferramenta *CSDiff* [7].

CSDiff é uma ferramenta de integração não-estruturada que se utiliza de um conjunto de separadores sintáticos das linguagens (símbolos utilizados para separar escopos e contextos) para realizar a análise sintática durante o processo de *merge*. A ideia do *CSDiff* é usar separadores sintáticos, além das quebras de linha, como divisores de contexto para detecção e resolução de conflitos sobre o *diff3*

https://pauloborba.cin.ufpe.br/publication/2024refinando_a_precisao_da_deteccao_de_conflitos__uma_analise_do_csdiff_com_abordagem_focalizada/

Less FPs and Fns

Métrica	SepInMerge	CSDiffI	SepMerge	CSDiff	diff3
Conflitos	329	726	323	537	373
Arquivos com conflitos	206	245	205	226	235
Cenários com conflitos	75	77	75	77	84

	SepMerge	diff3
aFP cenários	0	10
aFN cenários	2	0

	SepInMerge	diff3
aFP cenários	0	10
aFN cenários	2	0

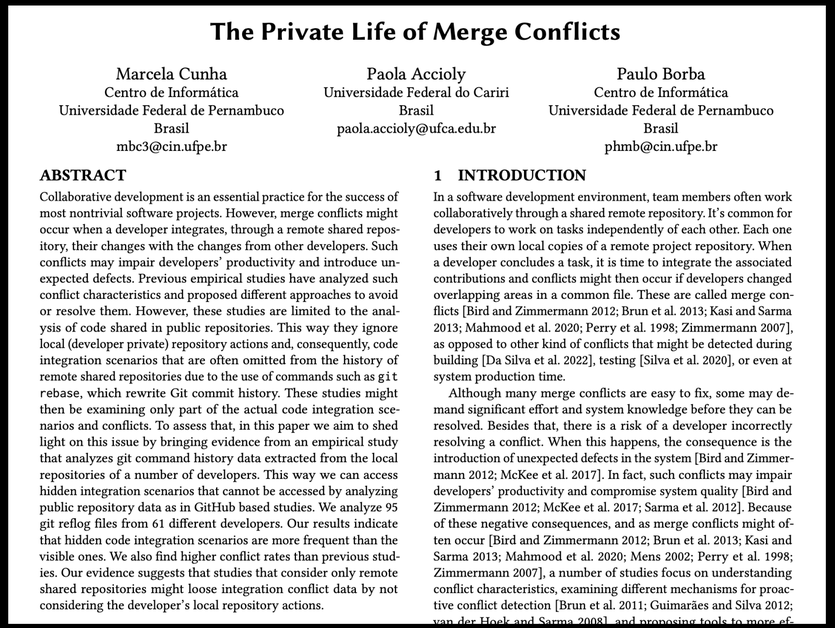
	CSDiff	diff3
aFP cenários	1	9
aFN cenários	2	0

	CSDiffI	diff3
aFP cenários	1	8
aFN cenários	3	0

Hidden integration scenarios are more frequent than the visible ones (13.3%) in the private repositories, representing 86.7%

Higher conflict rates in private repos studies

https://pauloborba.cin.ufpe.br/publication/2022the_private_life_of_merge_conflicts/



The Private Life of Merge Conflicts

Marcela Cunha
Centro de Informática
Universidade Federal de Pernambuco
Brasil
mbc3@cin.ufpe.br

Paola Accioly
Universidade Federal do Cariri
Brasil
paola.accioly@ufca.edu.br

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco
Brasil
phmb@cin.ufpe.br

ABSTRACT

Collaborative development is an essential practice for the success of most nontrivial software projects. However, merge conflicts might occur when a developer integrates, through a remote shared repository, their changes with the changes from other developers. Such conflicts may impair developers' productivity and introduce unexpected defects. Previous empirical studies have analyzed such conflict characteristics and proposed different approaches to avoid or resolve them. However, these studies are limited to the analysis of code shared in public repositories. This way they ignore local (developer private) repository actions and, consequently, code integration scenarios that are often omitted from the history of remote shared repositories due to the use of commands such as `git rebase`, which rewrite Git commit history. These studies might then be examining only part of the actual code integration scenarios and conflicts. To assess that, in this paper we aim to shed light on this issue by bringing evidence from an empirical study that analyzes git command history data extracted from the local repositories of a number of developers. This way we can access hidden integration scenarios that cannot be accessed by analyzing public repository data as in GitHub based studies. We analyze 95 git reflog files from 61 different developers. Our results indicate that hidden code integration scenarios are more frequent than the visible ones. We also find higher conflict rates than previous studies. Our evidence suggests that studies that consider only remote shared repositories might loose integration conflict data by not considering the developer's local repository actions.

1 INTRODUCTION

In a software development environment, team members often work collaboratively through a shared remote repository. It's common for developers to work on tasks independently of each other. Each one uses their own local copies of a remote project repository. When a developer concludes a task, it is time to integrate the associated contributions and conflicts might then occur if developers changed overlapping areas in a common file. These are called merge conflicts [Bird and Zimmermann 2012; Brun et al. 2013; Kasi and Sarma 2013; Mahmood et al. 2020; Perry et al. 1998; Zimmermann 2007], as opposed to other kind of conflicts that might be detected during building [Da Silva et al. 2022], testing [Silva et al. 2020], or even at system production time.

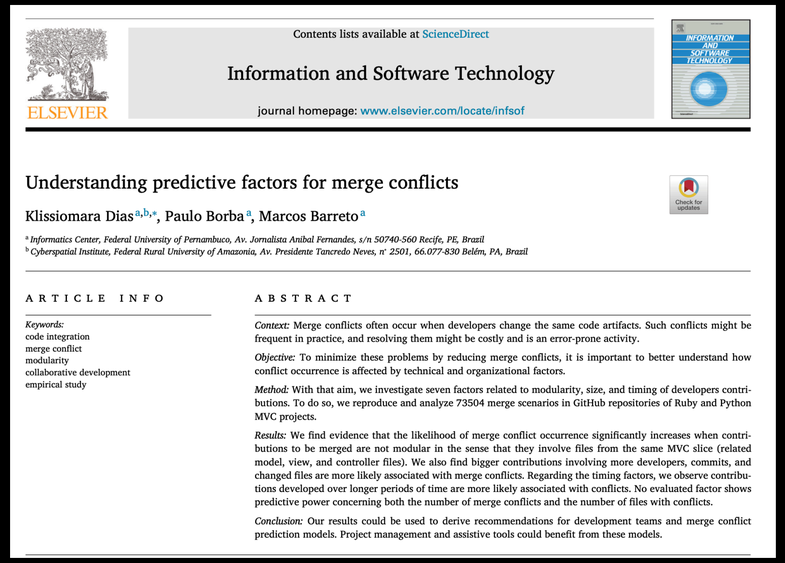
Although many merge conflicts are easy to fix, some may demand significant effort and system knowledge before they can be resolved. Besides that, there is a risk of a developer incorrectly resolving a conflict. When this happens, the consequence is the introduction of unexpected defects in the system [Bird and Zimmermann 2012; McKee et al. 2017]. In fact, such conflicts may impair developers' productivity and compromise system quality [Bird and Zimmermann 2012; McKee et al. 2017; Sarma et al. 2012]. Because of these negative consequences, and as merge conflicts might often occur [Bird and Zimmermann 2012; Brun et al. 2013; Kasi and Sarma 2013; Mahmood et al. 2020; Mens 2002; Perry et al. 1998; Zimmermann 2007], a number of studies focus on understanding conflict characteristics, examining different mechanisms for proactive conflict detection [Brun et al. 2011; Guimarães and Silva 2012; van der Hoek and Sarma 2008], and proposing tools to more ef-

The likelihood of merge conflict occurrence significantly increases when contributions to be merged are not modular in the sense that they involve files from the same MVC slice.

Bigger contributions involving more developers, commits, and changed files are more likely associated with merge conflicts.

Contributions developed over longer periods of time are more likely associated with conflicts.

No evaluated factor shows predictive power concerning both the number of merge conflicts and the number of files with conflicts.



Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Understanding predictive factors for merge conflicts

Klissiomara Dias[a,b,*], Paulo Borba[a], Marcos Barreto[a]

[a] Informatics Center, Federal University of Pernambuco, Av. Jornalista Aníbal Fernandes, s/n 50740-560 Recife, PE, Brazil
[b] Cyberspatial Institute, Federal Rural University of Amazonia, Av. Presidente Tancredo Neves, n° 2501, 66.077-830 Belém, PA, Brazil

ARTICLE INFO

Keywords:
code integration
merge conflict
modularity
collaborative development
empirical study

ABSTRACT

Context: Merge conflicts often occur when developers change the same code artifacts. Such conflicts might be frequent in practice, and resolving them might be costly and is an error-prone activity.

Objective: To minimize these problems by reducing merge conflicts, it is important to better understand how conflict occurrence is affected by technical and organizational factors.

Method: With that aim, we investigate seven factors related to modularity, size, and timing of developers contributions. To do so, we reproduce and analyze 73504 merge scenarios in GitHub repositories of Ruby and Python MVC projects.

Results: We find evidence that the likelihood of merge conflict occurrence significantly increases when contributions to be merged are not modular in the sense that they involve files from the same MVC slice (related model, view, and controller files). We also find bigger contributions involving more developers, commits, and changed files are more likely associated with merge conflicts. Regarding the timing factors, we observe contributions developed over longer periods of time are more likely associated with conflicts. No evaluated factor shows predictive power concerning both the number of merge conflicts and the number of files with conflicts.

Conclusion: Our results could be used to derive recommendations for development teams and merge conflict prediction models. Project management and assistive tools could benefit from these models.

https://pauloborba.cin.ufpe.br/publication/2020understanding_predictive_factors_for_merge_conflicts/

Textual merge, unstructured merge

Merge and Code Review

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br