Universidade Federal de Pernambuco
Centro de Informática

Bacharelado em Ciência da Computação

Análise de extensão do CSDiff para uso em linguagens com poucos separadores sintáticos

José Gabriel Silva Pereira

Trabalho de Graduação

Recife
27 de Abril de 2023

Universidade Federal de Pernambuco
Centro de Informática

José Gabriel Silva Pereira

Análise de extensão do CSDiff para uso em linguagens com poucos separadores sintáticos

Trabalho apresentado ao Programa de Bacharelado em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Paulo Henrique Monteiro Borba*
Co-orientadora: *Paola Rodrigues de Godoy Accioly*

Recife
27 de Abril de 2023

Resumo

A prática de desenvolvimento de software, há muito tempo deixou de ser uma tarefa que era realizada por somente uma pessoa, pois, com o avanço da tecnologia, sistemas cada vez mais complexos foram sendo criados fazendo com que muitas pessoas trabalhassem no mesmo projeto. Por conta disso, ferramentas de controle de versionamento de código foram criadas, permitindo que múltiplos desenvolvedores trabalhassem modificando o mesmo trecho de código simultaneamente. Porém, essas modificações simultâneas podem gerar conflitos quando feitas em um mesmo pedaço de código, o que impacta negativamente na produtividade de um time. Ao decorrer do tempo, diversas formas de como detectar conflitos na junção de versão de códigos foram criadas, dentre elas: linha a linha, estruturada e semiestruturada. Neste trabalho, é proposto uma extensão para uma ferramenta semiestruturada de detecção de conflitos ja existente, o *CSDiff* [1], de forma que ela utilize indentação como separador da linguagem, permitindo assim que, durante a detecção de conflitos em linguagens com poucos separadores sintáticos, ainda seja possivel permitir uma redução de falsos conflitos, consequentemente melhorando a produtividade de um time.

Palavras-chave: Processo de Merge, Desenvolvimento Colaborativo, Merge Textual, Merge Estruturado, Separadores sintáticos

Abstract

The practice of software development has long ceased to be a task performed by only one person, as with the advancement of technology, increasingly complex systems have been created, causing many people to work on the same project. Therefore, code version control tools were created, allowing multiple developers to work on modifying the same piece of code simultaneously. However, these simultaneous modifications can generate conflicts when made on the same piece of code, negatively impacting a team's productivity. Over time, several ways of detecting conflicts in the merging of code versions have been created, including line-by-line, structured, and semi-structured. In this work, an extension is proposed for an existing semi-structured conflict detection tool, the *CSDiff* [1], so that it uses indentation as a language separator, thus allowing a reduction in false conflicts during conflict detection in languages with few syntactic separators, consequently improving a team's productivity.

Keywords: Merge Process, Collaborative Development, Textual Merge, Structured Merge, Syntactic Separators.

CAPÍTULO 1

Introdução

Com o crescimento da complexidade dos sistemas de software, surge a necessidade de que múltiplas pessoas trabalhem num mesmo projeto. Essas modificações, com o objetivo de trazer mais produtividade, costumam ser executadas de forma paralela e podem acontecer em trechos de código em comum. Tendo como objetivo auxiliar os desenvolvedores a controlar e versionar suas modificações no código, ferramentas de controle de versão de código foram criadas. Essas ferramentas auxiliam a reduzir o trabalho extra quando se trata de modificações paralelas que precisam ser unidas. O processo de unir duas modificações paralelas de código é chamado de *merge* [2].

No processo de *merge*, quando dois desenvolvedores modificam o mesmo trecho de código e essas mudanças interferem uma na outra, é gerado um conflito. Esses conflitos, quando detectados, algumas vezes precisam ser resolvidos por um ou ambos os desenvolvedores, o que acaba impactando na produtividade, dado que resolvê-los geralmente é uma tarefa que geralmente demanda tempo [3]. Além do impacto na produtividade do time, quando esses conflitos não são detectados pela ferramenta de *merge*, ou quando são detectados e mal resolvidos, eles podem levar à introdução de bugs dentro do código, o que influencia na qualidade do produto final [4].

A abordagem de *merge* mais utilizada na indústria atualmente é o *merge* não estruturado [5], que se utiliza de uma análise puramente textual, equiparando linha a linha trechos do código para detectar e resolver conflitos. Porém, por não utilizar a estrutura do código que está sendo integrado, por muitas vezes essa abordagem gera falsos conflitos. Ao observar isso, pesquisadores propuseram ferramentas que se baseiam na estrutura dos arquivos que estão sendo integrados, criando uma árvore sintática a partir do texto dos arquivos e de sua linguagem de programação [6]. Essas abordagens são chamadas estruturadas e semiestruturadas.

Estudos anteriores [6]–[8] compararam as duas abordagens (estruturada e semiestruturada) em relação à não estruturada e mostraram que, para a maioria das situações de *merge* dos projetos, houve uma redução de conflitos em favor da semi ou da estruturada. Essa redução se dá por conta de falsos conflitos que possuem resolução óbvia, como por exemplo, quando os desenvolvedores adicionam dois métodos diferentes e independentes numa mesma região do código [9].

Esse benefício advém da exploração da estrutura gerada pela análise sintática, também chamada de análise gramatical. Ela envolve o agrupamento dos tokens (palavras) do programa fonte em frases. Cada linguagem possui conjuntos de tokens, onde alguns servem como divisores de elementos sintáticos e escopo semântico, como por exemplo as chaves (’’, ’’) numa linguagem como Java. Estes tokens, especificamente, são definidos aqui simplesmente como **separadores** sintáticos.

A solução não estruturada mais utilizada, o Diff3, se baseia somente na quebra de linha como o divisor de contexto para detecção de conflitos. Assim, o algoritmo de *merge* compara as linhas mantidas, adicionadas, e removidas por cada desenvolvedor e, com base nisso, reporta conflito quando as mudanças ocorrem em uma mesma área do texto, isto é, quando não há uma linha mantida que separa as mudanças feitas por um desenvolvedor das mudanças feitas pelo outro.

Como forma de melhorar os resultados do Diff3, o CSDiff, proposto em trabalhos anteriores, utiliza-se dos separadores mencionados acima para dividir o contexto de cada linha de código. Assim, o algoritmo de *merge* consegue, por exemplo, resolver conflitos em uma mesma linha, contanto que esses conflitos estejam separados por pelo menos um dos separadores definidos.

Contudo, o CSDiff possui limitações, pois linguagens como Python, possuem poucos separadores (seu principal separador é a própria indentação do código, que não é considerado pelo CSDiff atual). Este trabalho propõe uma modificação para o CSDiff, que utiliza a indentação como um separador sintático, de forma a tentar resolver esse problema, e analisa os resultados em comparação ao *merge* não estruturado puramente textual. Em particular, investiga-se as seguintes perguntas de pesquisa:

1) PP1: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de conflitos reportados em comparação ao *merge* puramente textual?
2) PP2: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de cenários com conflitos reportados em comparação ao *merge* puramente textual?
3) PP3: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de falsos conflitos e cenários com falsos conflitos reportados (falsos positivos) em comparação ao *merge* puramente textual?
4) PP4: A nova solução de *merge* não estruturado, utilizando indentação, amplia a possibilidade de comprometer a corretude do código, por aumentar o número de integrações de mudanças que interferem uma na outra, sem reportar conflitos (falsos negativos), além de aumentar cenários com falsos negativos?
5) PP5: A nova solução de *merge* não estruturado, utilizando indentação, demonstra um aumento de produtividade considerando o ato de resolver conflitos de merge?

Os resultados obtidos mostram que além de aumentar a quantidade de conflitos reportados (como esperado considerando os resultados dos trabalhos anteriores), o *merge* não estruturado utilzando separadores e indentação, demonstra, para a amostra utilizada, um aumento de aFN proporcional a quantidade de aFP reduzido quando comparado ao Diff3. Vale ressaltar que isso foi observado considerando a amostra sem um dos projetos utilizados - o matplotlib - que por ter uma quantidade muito grande de conflitos por cenário/arquivo, se tornou um ponto fora da curva ao considerar criação de aFP. Por outro lado, ao se fazer a análise de aumento de produtividade considerando o ato de resolver e reduzir conflitos, além de considerar geração de conflitos extras e resolução errada de conflitos, notou-se um bom aumento de produtividade com a utilização do CSDiff. Os conceitos utilizados para esta análise são explicados no capítulo (TODO: adicionar capitulo/secao aqui).

CAPÍTULO 2

Motivação

2.1 Merge Não Estrurado

Apesar da evolução dos sistemas de controle de versão, as ferramentas utilizadas para realizar *merge* não evoluiram tanto. A ferramenta mais utilizada atualmente é o *diff3* [2], que consiste em uma abordagem puramente textual baseada em linhas. Essa ferramenta, ao receber os três arquivos que configuram o cenário de merge (chamemos de *base*, *left* e *right*), compara, linha a linha, as duas versões modificadas (*left* e *right*) com seu ancestral comum (*base*), que foi o arquivo que deu origem às modificações que vão ser integradas. APós isso, a ferramenta agrupa as maiores áreas em comum e checa se existem interseções entre as áreas modificadas por *left* e *right*. Essas interseções são definidas como conflitos de merge [5], elas serão sinalizadas pelo *diff3* através de marcadores `(">>>>>>>","=======","<<<<<<<")`, como demonstrado na Figura 2.4.

Por considerar as mudanças linha a linha para detectar os conflitos, muitas vezes essa ferramenta relata falsos conflitos de modificações que alteram a mesma linha ou linhas consecutivas, mas que não interferem entre si semanticamente. Esse problema poderia ser resolvido utilizando ferramentas que explorem a estrutura sintática do código em questão.

Para ilustrar esse problema causado pelo *merge* não estruturado, utilizamos a implementação de um método to_string que recebe uma lista de strings e retorna uma string juntando todos os elementos da lista separados por dois *underlines* "__". Observe que as mudanças feitas pelo *left* foram a adição de uma nova condição no if, e a mudançã da string separadora do return final (modificado de ".."para "__"). Por outro lado, as mudanças feitas pelo *right* foram o valor padrao de retorno de uma string vazia para uma constante (por simplicidade consideramos que a constante foi definida na classe onde o método está sendo implementado), e a mesma modificação que o *left* fez no último return do método. Perceba que todas essas mudanças são independentes entre si, mas por elas acontecerem em linhas consecutivas, o *diff3* relata todas elas em um mesmo conflito (Figura 2.4).

```
1 def to_string(l: List[str]) -> str:
2     if len(l) == 0:
3         return ""
4     return "..".join(l)
```

Figura 2.1 Arquivo *base* que contém o método to_string

```
def to_string(l: List[str]) -> str:
    if l is null or len(l) == 0:
        return ""
    return "__".join(l)
```

Figura 2.2 Arquivo *left* que contém o método to_string

```
def to_string(l: List[str]) -> str:
    if len(l) == 0:
        return self.D
    return "__".join(l)
```

Figura 2.3 Arquivo *right* que contém o método to_string

2.2 Merge Semiestruturado e Estruturado

Como alternativa ao uso de merge não estruturado, existem as abordagens semiestruturadas ou completamente estruturadas. Ao contrário da abordagem não estruturada, essas abordagens levam em consideração a estrutura sintática da linguagem de programação para identificar conflitos com maior precisão e resolvê-los de forma mais correta. Essas abordagens criam árvores sintáticas para cada versão dos arquivos a serem integrados (*base*, *left* e *right*) e comparam essas árvores para identificar nós comuns e adições ou remoções em cada árvore. Dessa forma, cada elemento sintático é representado em nós distintos, e conflitos são sinalizados quando as mudanças a serem integradas estão relacionadas ao mesmo nó da árvore. Isso significa que, em vez de usar linhas como a unidade básica para comparação, essas ferramentas usam nós sintáticos como unidade.

Essas ferramentas estruturadas e semiestruturadas conseguem evitar falsos conflitos encontrados na abordagem não estruturada. Por exemplo, duas situações em que dois desenvolvedores adicionam separadamente dois novos métodos com diferentes assinaturas em uma mesma área do texto podem ser conciliadas com sucesso. As mudanças ocorrem na mesma linha, mas cada declaração é representada por um nó diferente, pois o identificador do método é parte do nó, e os dois nós são mantidos na árvore resultante da integração.

Dessa forma, é fácil observar que uma ferramenta estruturada para Python evitaria o conflito apresentado na Figura 2.4. A ferramenta, utilizando a estrutura da linguagem, identificaria que, apesar das mudanças representadas na Figura 2.2 e na Figura 2.3 ocorrerem em linhas consecutivas (o que faz com que o *diff3* agrupe as mudanças em um único bloco de conflito), elas estariam associadas a nós diferentes na árvore sintática. A ferramenta então juntaria as mudanças em uma versão resultante que contém a nova condição proposta por *left* e o novo valor de retorno proposto por *right*, evitando o falso conflito.

```
def to_string(l: List[str]) -> str:
<<<<<<< ./left.py
    if l is null or len(l) == 0:
        return ""
    return "__".join(l)
=======
    if len(l) == 0:
        return self.D
    return "__".join(l)
>>>>>>> ./right.py
```

Figura 2.4 Resultado de executar o *diff3*

2.3 Merge não Estruturado com separadores

As abordagens estruturadas e semiestruturadas discutidas na seção anterior, apesar de claramente benéficas, possuem um custo adicional, dado que elas se baseiam em manipulação de árvores sintáticas, que por sua vez são dependentes da linguagem em questão e demandam um esforço significativo de implementação por linguagem, além de que a manipulação das árvores sintáticas possui um custo computacional maior em relação a abordagens puramente textuais.

Em trabalho anterior, Clementino, Borba e Cavalcanti propõe uma nova ferramenta chamada **Custom Separators Diff** (*CSDiff*), cujo funcionamento baseia-se na abordagem puramente textual, mas que também considera a estrutura sintática do programa por meio de um conjunto de separadores da linguagem, escolhidos pelo usuário e passados como parâmetro.

(1) Transforma os arquivos *base*, *left* e *right*, fazendo com que os separadores sintáticos dados como entrada fiquem em linhas separadas; essas novas linhas são marcadas com a sequencia de caracteres: '$$$$$$$$';
(2) Chama o *merge* textual do Diff3, passando como entrada os arquivos gerados por 1;
(3) No arquivo resultante do Passo 2, remove as linhas extras e marcadores adicionados no Passo 1.

Figura 2.5 Processo do *CSDiff*

Para ilustrar o funcionamento do *CSDiff* e seu potencial para resolver falsos conflitos, observamos as figuras 2.6, 2.7 e 2.8 onde executamos o processo descrito na figura 2.5 utilizando os separadores sintáticos de Python "(", ")"e ",".

Nota-se na Figura 2.9 que ao executar o *CSDiff* nos arquivos *base*, *left* e *right*, as linhas conflitantes ficam mais separadas pelos marcadores (as sequencias "$$$$$$$$"), o que faz com que o conflito relatado pelo *diff3* seja reduzido (indicando que parte dele foi parcialmente resolvido automaticamente).

Entretanto, nota-se uma limitação para este caso na linguagem Python, visto que as duas expressões de return acabam não se separando simplesmente pelo fato de que não há nenhum

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
    if len
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
        return ""
    return "..".join
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 2.6 Arquivo *base* após a o Passo 1 da Figura 2.5

separador sintático entre eles. Essa limitação não ocorre em linguagens como Java, por exemplo, pois geralmente essas duas expressões return seriam separadas por um "}"(que em Java delimita final de escopo). Isso acontece para a linguagem Python pelo fato de que a mesma não possui separador para delimitar escopo, pois utiliza da indentação para tal.

Dada esta limitação, propomos neste trabalho uma modificação no algoritmo do *CSDiff*, visando resolver esse problema de separação de escopo. Para fazer isso, modificamos o algoritmo para que também adicione marcadores ao encontrar mudanças na indentação do programa.

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
    if l is null or len
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
        return ""
    return "__".join
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 2.7 Arquivo *left* após a o Passo 1 da Figura 2.5

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
    if len
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
        return self.D
    return "__".join
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 2.8 Arquivo *right* após a o Passo 1 da Figura 2.5

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
    if l is null or len
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
<<<<<<< left.py_temp.py
        return ""
    return "__".join
=======
        return self.D
    return "__".join
>>>>>>> right.py_temp.py
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 2.9 Arquivo resultante após a execução do Passo 2 da Figura 2.5 nos arquivos *base*, *left* e *right*

CAPÍTULO 3

Solução

3.1 Visão Geral da Implementação

3.1.1 Otimização e Simplificação com uso de *awk*

Primeiramente, para que consiguíssemos fazer o *CSDiff* detectar mudanças de indentação, era necessário que o programa iterasse linha a linha, no Passo 1 da Figura 2.5. Para tal, foi escolhido a ferramenta *awk* [10], utilizada primeiramente para simplificar alguns dos scripts feitos com a ferramenta *sed* que foram utilizados em [1], [11]. Esta simplificação utilizou-se do conceito chamado de *charclass* (add footnote: https://www.regular-expressions.info/charclass.html), que permite criar um comando de substituição mais simples do que o utilizado anteriormente, que consistia em uma sequencia de substituições *sed* individuais. Essa simplificação foi posteriormente testada pelo autor utilizando o *miningframework* (utilizado nessa pesquisa, bem como nas anteriores já mencionadas), para comprovar sua equivalência em relação ao script inicial (sequência de comandos *sed*). Além disso, apesar de não ter sido metodologicamente analisada (por não fazer parte do escopo desta pesquisa), notou-se uma melhora na velocidade de execução do *CSDiff* de aproximadamente 3x, indicando uma diferença considerável de performance entre as ferramentas *awk* e *sed*.

3.1.2 Inserindo Marcadores ao Detectar Mudanças de Indentação

Após a simplicicação acima, foi criada uma nova versão do *CSDiff* cujo processo é descrito na Figura 3.1 e seu resultado ilustrada nas Figuras 3.2, 3.3, 3.4, enquanto que o resultado da execução do *diff3* pode ser visto na figura 3.5

(1) Iterando linha a linha nos arquivos *base*, *left* e *right*:
   (a) Transforma os arquivos, fazendo com que os separadores sintáticos dados como entrada fiquem em linhas separadas; essas novas linhas são marcadas com a sequencia de caracteres: '$$$$$$$$', e adiciona uma linha de marcadores a mais acima da linha atual sempre que a linha anterior está em um nível de indentação diferente da linha atual.
(2) Chama o *merge* textual do Diff3, passando como entrada os arquivos gerados por 1;
(3) No arquivo resultante do Passo 2, remove as linhas extras e marcadores adicionados no Passo 1.

Figura 3.1 Processo do *CSDiff*

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
$$$$$$$
    if len
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
$$$$$$$
        return ""
$$$$$$$
    return "..".join
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 3.2 Arquivo *base* após a o Passo 1 da Figura 3.1

Com essas modificações, fizemos o experimento descrito na seção seguinte.

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
$$$$$$$
    if l is null or len
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
$$$$$$$
        return ""
$$$$$$$
    return "__".join
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 3.3 Arquivo *left* após a o Passo 1 da Figura 3.1

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
$$$$$$$
    if len
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
$$$$$$$
        return self.D
$$$$$$$
    return "__".join
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 3.4 Arquivo *right* após a o Passo 1 da Figura 3.1

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
$$$$$$$
    if l is null or len
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
$$$$$$$
        return self.D
$$$$$$$
    return "__".join
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 3.5 Arquivo resultante após a execução do Passo 2 da Figura 3.1 nos arquivos *base*, *left* e *right*. Note que agora o diff3 conseguiu resolver todos os conflitos de forma automática.

```
def to_string(l: List[str]) -> str:
    if l is null or len(l) == 0:
        return self.D
    return "__".join(l)
```

Figura 3.6 Arquivo resultante após a o Passo 3 da Figura 3.1

CAPÍTULO 4

Avaliação

Para avaliar esta nova versão do *CSDiff* e responder as perguntas de pesquisa mencionadas anteriormente, repetimos os experimentos feitos em [1], [11], que compara os resultados de utilizar *CSDiff* com o resultado de utilizar *diff3*, analisando o potencial para resolução de conflitos sem gerar impacto negativo na corretude do processo de *merge*.

4.1 CONCEITOS

A seguir, alguns conceitos relevantes para a avaliação serão definidos.

4.1.1 Cenário de Merge

Em um sistema de controle de versão, um *commit* é uma versão que agrupa mudanças em determinados arquivos de um projeto [12]. Considerando essa definição, um Cenário de Merge é definido como uma quádrupla de *commits*, que chamaremos aqui de *base*, *left*, *right* e *merge*. O *base* representa o *commit* de onde as modificações *left* e *right* partiram, enquanto o *merge* representa a versão final onde a integração das mudanças foi feita no repositório.

4.1.2 Falso Positivo Adicionado

Seguindo as mesmas definições descritas em [1], um falso positivo ocorre quando a ferramenta de merge relata um conflito que na verdade não deveria ter ocorrido, ou seja, as mudanças que estão sendo integradas não interferem uma na outra. Para comparar duas ferramentas de merge X e Y, usamos o conceito de Falso Positivo Adicionado *aFP*, que acontece quando a ferramenta X relata conflito em um determinado cenário de merge, enquanto a ferramenta Y não relata conflito no mesmo cenário e as mudanças integradas pelas ferramentas não interferem uma na outra. É importante usar o conceito de aFP porque o conjunto de falsos conflitos de uma ferramenta não é um subconjunto da outra.

4.1.3 Falso Negativo Adicionado

Um falso negativo ocorre quando a ferramenta de merge não relata um conflito que deveria ter sido reportado, pois as mudanças que estão sendo integradas interferem uma na outra. Ao comparar duas ferramentas de merge X e Y, usamos o conceito de Falso Negativo Adicionado (*aFN*), que acontece quando a ferramenta X não relata conflito em um determinado cenário de merge, enquanto a ferramenta Y relata conflito no mesmo cenário e as mudanças que estão

sendo integradas pelas ferramentas interferem uma na outra (ou seja, deveria ocorrer conflito). É importante usar o conceito de aFN porque o conjunto de falsos negativos de uma ferramenta não é um subconjunto da outra.

4.2 PERGUNTAS DE PESQUISA

A avaliação desta pesquisa é baseada em responder as seguintes perguntas de pesquisa.

4.2.1 PP1: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de conflitos reportados em comparação ao *merge* puramente textual?

Para avaliar o número de conflitos gerados pelo *diff3* e *CSDiff*, contamos o número de conflitos em cada ferramenta para cada cenário de merge. Para isso, executamos a ferramenta para os conjuntos de arquivos de *left*, *right* e *base* em cada cenário, resultando em um conjunto de arquivos combinados. Em seguida, contamos a ocorrência de marcadores de conflito para cada arquivo presente nesses conjuntos, que são sequências de caracteres apresentados no formato de conflito descrito na Figura 2.4.

4.2.2 PP2: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de cenários com conflitos reportados em comparação ao *merge* puramente textual?

Para avaliar o número de cenários de merge com conflitos gerados pelo *diff3* e *CSDiff*, contamos o número de cenários em cada ferramenta em que houve conflito. Para isso, executamos a ferramenta para os conjuntos de arquivos de *left*, *right* e *base* em cada cenário de merge, resultando em um conjunto de arquivos combinados. Um cenário é considerado com conflito se pelo menos um arquivo no conjunto resultante do merge tiver um conflito.

4.2.3 PP3: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de falsos conflitos e cenários com falsos conflitos reportados (falsos positivos) em comparação ao *merge* puramente textual?

Para responder a esta pergunta, foram contabilizados os casos em que uma ferramenta apresentou um falso positivo adicionado (*aFP*) em relação à outra. Para verificar se um cenário continha um *aFP* do *diff3* em comparação com o *CSDiff*, foi verificado se o *diff3* retornou pelo menos um conflito em pelo menos um dos arquivos integrados no cenário, enquanto o *CSDiff* não retornou nenhum conflito para todos os arquivos do cenário e obteve o resultado correto do merge. Neste caso, foi contabilizado que o *diff3* tinha um cenário com *aFP* em relação ao *CSDiff*. O mesmo procedimento foi realizado para encontrar *aFPs* adicionados pelo *CSDiff* em comparação com o *diff3*.

4.2.4 PP4: A nova solução de *merge* não estruturado, utilizando indentação, amplia a possibilidade de comprometer a corretude do código, por aumentar o número de integrações de mudanças que interferem uma na outra, sem reportar conflitos (falsos negativos), além de aumentar cenários com falsos negativos?

Para verificar se um cenário possui um *aFN* para uma ferramenta em comparação com outra, o merge foi executado usando tanto o *diff3* quanto o *CSDiff*. Se o *CSDiff* não retornasse nenhum conflito em nenhum arquivo resultante do merge no conjunto de arquivos do cenário, enquanto o *diff3* retornasse conflito e o *CSDiff* falhasse na integração das mudanças, esse cenário seria contado como um *aFN* para o *CSDiff*. Falhar na integração significa que a ferramenta resultou em um código sintaticamente incorreto ou que não preserva os comportamentos esperados individualmente pelas mudanças de *left* e *right*. Esse procedimento também foi realizado para encontrar falsos negativos adicionados pelo *diff3* em comparação com o *CSDiff*. Para verificar esses casos de falsos negativos, os códigos foram analisados manualmente.

4.2.5 PP5: A nova solução de *merge* não estruturado, utilizando indentação, demonstra um aumento de produtividade considerando o ato de resolver conflitos de merge?

Além das 4 primeiras perguntas de pesquisa, foi criada uma outra forma de analisar os benefícios do CSDiff em relação ao diff3. Para tal, foram definidas as 4 situações abaixo e cada situação foi associada a uma pontuação, de forma que uma maior pontuação em um determinado arquivo ou cenário, indica um aumento na produtividade do desenvolvedor ao utilizar o CSDiff como ferramenta ao invés do diff3. Essa análise foi feita manualmente comparando para cada arquivo de cada cenário de merge, os conflitos relatados pela ferramenta CSDiff e diff3, bem como foi comparado seus resultados com o resultado final do merge, nos casos onde um deles não relatava conflitos. Essa análise foi possível pois, apesar de a amostra ter sido consideravelmente grande (mais de 3000 cenários de merge), houveram aproximadamente 30 cenários onde tais comportamentos fossem possíveis de acontecer. Isso será melhor explicado na seção 4.3.

4.2.5.1 Conflito Reduzido

Um Conflito Reduzido é definido como um conflito que ocorre na ferramenta diff3 e na ferramenta CSDiff, mas que no resultado do CSDiff esse conflito está consideravelmente reduzido (possui um tamanho menor). Dessa forma, a pontuação escolhida para o aumento de produtividade nessa situação foi +1, dado que uma redução no tamanho de um conflito implica numa resolução mais rápida pelo desenvolvedor.

4.3 METODOLOGIA

4.4 RESULTADOS

4.4.1 PP1: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de conflitos reportados em comparação ao *merge* puramente textual?

4.4.2 PP2: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de cenários com conflitos reportados em comparação ao *merge* puramente textual?

4.4.3 PP3: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de falsos conflitos e cenários com falsos conflitos reportados (falsos positivos) em comparação ao *merge* puramente textual?

4.4.4 PP4: A nova solução de *merge* não estruturado, utilizando indentação, amplia a possibilidade de comprometer a corretude do código, por aumentar o número de integrações de mudanças que interferem uma na outra, sem reportar conflitos (falsos negativos), além de aumentar cenários com falsos negativos?

4.4.5 PP5: A nova solução de *merge* não estruturado, utilizando indentação, demonstra um aumento de produtividade considerando o ato de resolver conflitos de merge?

4.5 DISCUSSÃO

4.6 AMEAÇAS A VALIDADE

CAPÍTULO 5

Trabalhos Relacionados

CAPÍTULO 6

Conclusão

Bibliografia

[1] J. Clementino, P. Borba e G. Cavalcanti, "Textual merge based on language-specific syntactic separators," em *35th Brazilian Symposium on Software Engineering (SBES 2021)*, 2021, pp. 243–252.

[2] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, v. 28, n. 5, pp. 449–462, 2002. DOI: 10.1109/TSE.2002.1000449.

[3] Y. Brun, R. Holmes, M. D. Ernst e D. Notkin, "Proactive Detection of Collaboration Conflicts," em *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, sér. ESEC/FSE '11, Szeged, Hungary: Association for Computing Machinery, 2011, pp. 168–178, ISBN: 9781450304436. DOI: 10.1145/2025113.2025139. endereço: https://doi.org/10.1145/2025113.2025139.

[4] C. Brindescu, I. Ahmed, C. Jensen e A. Sarma, "An empirical investigation into merge conflicts and their effect on software quality," *Empirical Software Engineering*, v. 25, n. 1, pp. 562–590, 2020.

[5] S. Khanna, K. Kunal e B. C. Pierce, "A formal investigation of diff3," em *International Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer, 2007, pp. 485–496.

[6] S. Apel, J. Liebig, B. Brandl, C. Lengauer e C. Kästner, "Semistructured Merge: Rethinking Merge in Revision Control Systems," em *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, sér. ESEC/FSE '11, Szeged, Hungary: Association for Computing Machinery, 2011, pp. 190–200, ISBN: 9781450304436. DOI: 10.1145/2025113.2025141. endereço: https://doi.org/10.1145/2025113.2025141.

[7] G. Cavalcanti, P. Accioly e P. Borba, "Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment," em *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2015, pp. 1–10. DOI: 10.1109/ESEM.2015.7321191.

[8] G. Cavalcanti, P. Borba, G. Seibt e S. Apel, "The Impact of Structure on Software Merging: Semistructured Versus Structured Merge," em *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1002–1013. DOI: 10.1109/ASE.2019.00097.

[9] G. Cavalcanti, P. Borba e P. Accioly, "Evaluating and Improving Semistructured Merge," *Proc. ACM Program. Lang.*, v. 1, n. OOPSLA, out. de 2017. DOI: 10.1145/3133883. endereço: https://doi.org/10.1145/3133883.

[10] A. Robbins, *Gawk: Effective AWK Programming*. Free Software Foundation Boston, 2004.

[11] H. S. C. Souza, *Extensão e análise de performance da ferramenta de merge textual CSDiff para novas linguagens*, Graduation Thesis, 2021.

[12] A. Koc e A. U. Tansel, "A survey of version control systems," *ICEME 2011*, 2011.