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

1 INTRODUÇÃO

Com o crescimento da complexidade dos sistemas de software, surge a necessidade de projetos serem executados por equipes formadas por várias pessoas trabalhando em conjunto e modificando artefatos de software em comum. Essas modificações, com o objetivo de trazer maior produtividade, muitas vezes são feitas simultaneamente, na mesma área de texto de cópias de artefatos armazenados em repositórios individuais dos desenvolvedores da equipe [12]. Quando essas modificações são integradas em um repositório comum à equipe, através de um processo e ferramenta de *merge* [17, 19], são detectados conflitos de *merge*. Tais conflitos precisam ser geralmente resolvidos por um ou ambos os desenvolvedores, o que acaba impactando na produtividade da equipe, dado que resolvê-los geralmente é uma tarefa tediosa e que demanda tempo [5, 7]. Além do impacto na produtividade, quando esses conflitos não são detectados pela ferramenta de *merge*, ou quando detectados e mal resolvidos, eles podem levar à introdução de *bugs* dentro do código, o que influencia negativamente na qualidade do produto final [6].

SBES '21, September 27-October 1, 2021, Joinville, Brazil

ACM ISBN 978-1-4503-9061-3/21/09…$15.00
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].

Esses benefícios, no entanto, têm um custo associado, já que essas ferramentas baseadas na manipulação de árvores sintáticas são dependentes de linguagem, e requerem um significativo esforço de implementação para cada linguagem de programação. Para evitar esse custo, propomos neste artigo um novo tipo de ferramenta textual de *merge*, que explora parte da estrutura sintática da linguagem, mas sem a necessidade de criar e manipular árvores. Enquanto as ferramentas (semi) estruturadas se baseiam na ocorrência de sequências de *tokens* específicos para determinar que tipo de nó criar na árvore sintática, nossa ferramenta se baseia na ocorrência de determinados caracteres para delimitar as áreas do texto a serem equiparadas para detecção de conflitos. Assim, ao invés de uma equiparação linha a linha (sequência de caracteres delimitada por '\n'), temos uma equiparação de sequências de caracteres delimitadas não só por '\n' mas também por *separadores de elementos sintáticos* e escopo como, por exemplo, as chaves ('{', '}') em uma linguagem como Java ou C. Fornecendo diferentes conjuntos de separadores como entrada para a ferramenta, pode-se lidar com linguagens diferentes sem custo extra de implementação para cada linguagem. Como a ferramenta não estruturada tradicional se baseia somente na quebra de linha como o divisor de contexto para detecção de conflitos, ela reporta conflito sempre que tenta integrar mudanças feitas numa mesma linha ou em linhas consecutivas. Contrastando, nossa ferramenta reporta conflito apenas quando tenta integrar mudanças feitas na mesma área de texto delimitada pelos separadores.

Além de apresentar a ideia desse novo tipo de ferramenta de *merge*, e discutir a sua implementação preliminar (CSDiff), neste artigo comparamos a sua acurácia em relação ao *merge* não estruturado textual tradicional (Diff3) [16]. Em particular, investigamos as seguintes perguntas de pesquisa (PP):

- **PP1** O CSDiff reduz a quantidade de conflitos reportados em comparação ao Diff3?
- **PP2** O CSDiff reduz a quantidade de cenários com conflitos reportados em comparação ao Diff3?
- **PP3** O CSDiff reduz a quantidade de falsos conflitos e cenários com falsos conflitos reportados (falsos positivos) em comparação ao Diff3?
- **PP4** O CSDiff, em comparação ao Diff3, aumenta o número de integrações de mudanças que interferem uma na outra sem reportar conflitos (falsos negativos)?

Os resultados obtidos para uma amostra de cenários de *merge* (cada cenário é uma quádrupla de *commits* formada por um *commit* de *merge*, seus dois pais, e o ancestral comum mais recente dos pais) extraídos de projetos Java do GitHub mostram que o CSDiff, o *merge* não estruturado utilizando separadores, reduz o número de *falsos positivos* e *cenários com conflitos* em comparação ao Diff3. Essa redução sugere que o CSDiff pode trazer benefícios para o processo de integração de código sem demandar os custos extras das ferramentas estruturadas e semiestruturadas. Isso motiva novos estudos e aprimoramentos da abordagem proposta aqui.

Por outro lado, na nossa amostra, o CSDiff aumenta consideravelmente a quantidade de *conflitos* reportados, por uma questão puramente de granulosidade dos conflitos; uma dada mudança conflitante que leva o Diff3 a reportar um único conflito pode fazer com que o CSDiff reporte vários conflitos para a mesma mudança. Ou seja, não é que o CSDiff sinalize, de forma significativa, conflitos em situações em que o Diff3 não sinaliza; é que, para a mesma situação, o CSDiff sinaliza vários conflitos menores ao invés de um único conflito maior. Isso também foi observado para ferramentas estruturadas em estudos anteriores [11]. Uma abordagem simples como executar o Diff3 toda vez que o CSDiff resultar em conflito, resolveria este problema, mantendo os conflitos de maior granulosidade do Diff3— e assim reduzindo o número de conflitos— ao mesmo tempo em que reduz a quantidade de falsos positivos e cenários com conflitos.

Já com relação a *falsos negativos*, o CSDiff apresenta um aumento quando comparado com o Diff3, mas proporcionalmente bem menor que a redução em falsos positivos. Por fim, nossa análise com diferentes conjuntos de separadores mostra que eles podem influenciar na acurácia da ferramenta, com mais separadores levando a maior redução de falsos positivos e maior aumento de falsos negativos.

2 MOTIVAÇÃO

2.1 *Merge* não estruturado

Apesar de os sistemas de controle de versão de código terem evoluído ao longo dos anos, as ferramentas utilizadas para realizar *merge* não evoluíram tanto. A estratégia mais utilizada atualmente é, e tem sido por muitos anos, a abordagem textual baseada em linhas. Mais especificamente um *merge não estruturado* que se utiliza do algoritmo **diff3** [19]. Ao receber três arquivos, o *merge* não estruturado compara, linha a linha, os dois arquivos modificados (vamos chamar de *left* e *right*) com relação ao ancestral em comum, o arquivo *base*, a versão do arquivo que deu origem às modificações feitas em separado e que vão ser integradas. A ferramenta agrupa as maiores áreas em comum (que não foram modificadas) de *left* e *right* em relação à *base*, e verifica se há interseções entre as áreas que *left* modificou com as que *right* modificou. Havendo interseção, a ferramenta sinaliza um conflito [16]; caso contrário, combina as modificações feitas em áreas distintas.

```
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`

Por utilizar apenas a análise das linhas para realizar a operação de integração das mudanças dos arquivos, o *merge* não estruturado acaba, em muitos casos, relatando falsos conflitos de modificações que alteram uma mesma linha de um arquivo, ou linhas consecutivas, mas que não são excludentes por não interferirem entre si; a semântica e a intenção de uma mudança não é afetada pela outra. Essa imprecisão [2, 9] poderia, em parte, ser evitada se a ferramenta explorasse a estrutura sintática do programa sendo integrado [18], como mostraremos mais adiante.

Para ilustrar esse problema de falsos conflitos causados pelo *merge* não estruturado, utilizamos uma implementação simples de um método `toString` que recebe uma lista e retorna uma *String*, com os elementos da lista separados por vírgulas. A versão *base* deste código, em Java, é apresentada na Figura 1. Por simplicidade, é admitido que este método é a única parte do arquivo que será modificada neste exemplo. Agora, considere que uma desenvolvedora, partindo de *base*, alterou a condição do `if` na linha 2 para verificar se a lista é nula ou vazia, obtendo a versão *left* na Figura 2. Independentemente, um desenvolvedor, partindo de *base*, modificou o código extraindo a constante `D` também na linha 2, levando ao arquivo *right* ilustrado na Figura 3.

Ao tentar integrar essas duas mudanças em um repositório comum da equipe, o integrador que usar o *merge* não estruturado tradicional[1] observará o alerta de conflito mostrado na Figura 4. O arquivo resultante mostra que, por as mudanças terem ocorrido na mesma linha do código, a ferramenta não foi capaz de resolver as incompatibilidades e integrar os arquivos com sucesso de forma a

[1]Por exemplo, com a ferramenta diffutils, que contém o *diff3*, chamado com a opção *-m* e os três arquivos apresentados.

```
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 }
```

Figure 4: Resultado de executar o *diff3*

gerar um arquivo que contém e concilia as mudanças feitas tanto em *left* quanto em *right*.

2.2 *Merge* estruturado e semiestruturado

Uma alternativa ao *merge* não estruturado é utilizar abordagens semiestruturadas ou completamente estruturadas. Diferentemente da abordagem não estruturada, essas abordagens exploram a estrutura sintática da linguagem de programação para conciliar mudanças de forma mais eficaz e identificar conflitos com maior acurácia. Elas criam árvores sintáticas para cada versão dos arquivos a serem integrados, e comparam essas árvores para identificar nós comuns e adicionados ou removidos em cada árvore. Dessa forma, cada elemento sintático é representado em nós diferentes, e conflitos são sinalizados quando as mudanças a serem integradas estão relacionadas ao mesmo nó da árvore. Assim, ao invés de usar linhas como a unidade básica para comparação, essas ferramentas usam nós sintáticos como unidade.

Dessa forma, as ferramentas estruturadas e semiestruturadas conseguem evitar falsos conflitos da abordagem não estruturada [1, 2, 9–11], conciliando com sucesso, por exemplo, uma situação em que dois desenvolvedores adicionam separadamente dois novos métodos, com diferentes assinaturas, numa mesma área do texto. Nesse caso, as mudanças ocorrem na mesma linha, mas cada declaração é representada por um nó diferente— já que o identificador do método é parte do nó— e os dois nós são mantidos na árvore resultante da integração.

De forma similar, uma ferramenta estruturada[2] para Java evitaria o conflito apresentado na Figura 4. A ferramenta, se utilizando da estrutura dessa linguagem, identifica que, apesar das mudanças representadas na Figura 2 e na Figura 3 ocorrerem na mesma linha, elas estão associadas a nós diferentes da árvore sintática. A ferramenta então concilia as mudanças em uma versão resultante que contém a nova condição proposta por *left* e a extração de constante proposta por *right*, evitando o falso conflito.

2.3 *Merge* não estruturado com separadores

Os benefícios discutidos na seção anterior, no entanto, têm um custo associado, já que essas ferramentas baseadas na manipulação de árvores sintáticas são dependentes de linguagem, e requerem um significativo esforço de implementação para cada linguagem de programação. Além disso, a criação, análise e manipulação de árvores sintáticas demandam um maior custo computacional em relação à uma análise puramente textual. Apesar desse custo adicional não ser proibitivo na maioria das situações de merge [2], ele precisa ser considerado.

Para investigar a possibilidade de eliminar esses custos extras sem comprometer parte dos benefícios das ferramentas estruturadas, propomos neste artigo um novo tipo de ferramenta textual de *merge*, que explora parte da estrutura sintática da linguagem, mas sem a necessidade de criar e manipular árvores. Enquanto as ferramentas (semi) estruturadas se baseiam na ocorrência de sequências de *tokens* específicos para determinar que tipo de nó criar na árvore sintática, a ideia básica da nossa ferramenta é se basear na ocorrência de determinados carácteres para delimitar as áreas do texto a serem equiparadas para detecção de conflitos. Assim, ao invés de uma equiparação linha a linha (sequência de carácteres delimitada por '\n'), temos uma equiparação de sequências de carácteres delimitadas não só por '\n' mas também por *separadores de elementos sintáticos* e escopo como, por exemplo, as chaves ('{', '}') em uma linguagem como Java.

Trabalhando dessa maneira, a nossa ferramenta evita o conflito apresentado na Figura 4. A nossa ferramenta considera '{' como um separador de áreas de textos, e assim identifica que as mudanças representadas na Figura 2 e na Figura 3 ocorrem em áreas diferentes, uma antes e outra após o '{' que temos na linha 2. De forma similar à ferramenta estruturada para Java que discutimos na seção anterior, a nossa ferramenta concilia as mudanças em uma versão resultante que contém a nova condição proposta por *left* e a extração de constante proposta por *right*, evitando o falso conflito.

3 SOLUÇÃO

A partir da ideia discutida na Seção 2.3, implementamos um protótipo de uma ferramenta não estruturada baseada em separadores sintáticos, chamada **Custom Separators Diff** (CSDiff).[3] A implementação tem como base o *merge* não estruturado, mais especificamente a ferramenta *diff3* com a opção de *merge* (*-m*), que chamaremos simplesmente Diff3. Apesar disso, simulamos parte do comportamento do estruturado, separando, tanto por quebras de linha quanto por separadores sintáticos, os trechos do código a serem integrados. Dessa forma, a ferramenta necessita receber como entrada os separadores sintáticos que serão utilizados para cada linguagem. Conhecendo os separadores, o CSDiff segue o processo descrito na Figura 5.

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

Figure 5: Processo do CSDiff

[2]Uma ferramenta semiestruturada para Java não evitaria o conflito, já que as mudanças do exemplo ocorrem no corpo de um método, que normalmente é tratado como uma *string* e integrado com uma ferramenta não estruturada; esse tipo de solução híbrida é o que justamente diferencia as ferramentas semiestruturadas das estruturadas.

[3]Link para o *protótipo*.

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

$$$$$$$}
$$$$$$$
```

Figure 6: Arquivo *base* temporário

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

$$$$$$$}
$$$$$$$
```

Figure 7: Arquivo *left* temporário

Para ilustrar o funcionamento do CSDiff, retomamos o exemplo da Seção 2.1, assumindo que o CSDiff recebeu como entrada o conjunto de separadores formado apenas pelos caracteres abre e fecha chave: '{' e '}'. Entre outros elementos sintáticos, na linguagem Java esses caracteres demarcam, respectivamente, o início e o fim de um bloco de código, e assim definem uma nova área de texto para efeito da análise feita pelo CSDiff. Como apresentado previamente, a desenvolvedora modificou a condição do `if` (Figura 2), enquanto que o desenvolvedor extraiu uma constante (Figura 3). Como observado na Figura 4, o Diff3 sinaliza um conflito pois não é capaz de integrar mudanças que ocorrem na mesma linha. Em contraste, por utilizar o caractere '{' como separador, o CSDiff identifica que as mudanças ocorrem em áreas diferentes e podem ser integradas sem conflito. Para entender melhor o funcionamento da ferramenta, ilustramos o seu funcionamento passo a passo.

No primeiro passo do processo, o CSDiff adiciona uma quebra de linha e o marcador '$$$$$$$' antes e depois de cada separador encontrado. Essa operação é realizada para cada arquivo passado para a ferramenta, gerando três arquivos temporários que podem ser observados nas Figuras 6 (*base*), 7 (*left*) e 8 (*right*).

Em seguida, no segundo passo do processo, o CSDiff executa o Diff3, tendo como entrada os arquivos temporários gerados pelo passo 1. O resultando gerado pelo Diff3 é exibido na Figura 9. Note que a integração é feita com sucesso pois a modificação de *left* é feita na linha 4, enquanto a de *right* é feita na linha 6. Com mudanças em linhas diferentes e não consecutivas, o Diff3 consegue realizar a integração, mantendo as linhas não alteradas por ambos, *left* e *right*, e incorporando as linhas modificadas por eles.

Finalmente, no último passo, o CSDiff remove do arquivo resultante do segundo passo todas as novas linhas criadas no passo 1 e marcadas com '$$$$$$$', gerando o seu resultado final, que pode ser observado na Figura 10. Em situações de conflito no segundo passo (por exemplo, caso *right* tivesse também alterado a condição do `if`), o processo é o mesmo e o CSDiff sinaliza o conflito sem as linhas extras e marcadores adicionados no passo 1.

O CSDiff é, portanto, uma ferramenta que, apesar de utilizar o *merge* não estruturado como base, em algumas situações consegue obter um resultado similar ao *merge* estruturado através de seu pré-processamento baseado em separadores sintáticos. Com isso, ele é capaz de evitar falsos conflitos apresentados pela abordagem puramente textual baseada em linhas.

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

$$$$$$$}
$$$$$$$
```

Figure 8: Arquivo *right* temporário

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

$$$$$$$}
$$$$$$$
```

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

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

Figure 10: Resultado final do CSDiff

Por ser ainda um protótipo, a versão atual do CSDiff apresenta algumas limitações. Dentre elas temos questões mais simples como a dependência de plataforma (Linux) e a não parametrização do

conjunto de separadores e marcadores[4], até questões que exigiriam um maior esforço de implementação, como a não identificação do contexto em que um separador ocorre. Por exemplo, arquivos com ocorrências de separadores em comentários Java ou constantes do tipo *String*— como `"set {2,4}"`— serão quebrados nessas áreas de texto também, quando o ideal seria evitar a quebra. Isso pode resultar em comportamentos inesperados para a ferramenta. Nesse caso, por exemplo, dois desenvolvedores alterariam a mesma constante, digamos antes e depois do '`{`', e a ferramenta não sinalizaria conflito.

4 AVALIAÇÃO

Para avaliar o CSDiff e responder as perguntas de pesquisa mencionadas anteriormente, analisamos o potencial do CSDiff para integração de código, comparando-o com o Diff3, uma das ferramentas mais utilizadas para integração de código na prática. Em particular, investigamos a acurácia das duas ferramentas de *merge* utilizando diversas métricas que capturam a capacidade de resolução de conflitos sem gerar impacto negativo na corretude do processo de *merge*; isso é essencial para avaliar se uma dada ferramenta pode ajudar a aumentar a produtividade de uma equipe de desenvolvimento de software sem comprometer a qualidade do produto sendo desenvolvido por essa equipe.

4.1 Visão geral da metodologia

Para comparar as duas ferramentas, primeiro mineramos *commits* de *merge* em projetos GitHub. Para cada *commit* de *merge*, identificamos o *cenário de merge* correspondente: uma quádrupla de *commits* formada pelo *commit* de *merge* (chamaremos de *rmerge*, para indicar que é o resultado do *merge* encontrado no *r*epositório do projeto), seus dois pais (aqui chamados de *left* e *right*), e o ancestral comum mais recente dos pais (*base*). Como cada cenário de *merge* possui quatro conjuntos de arquivos, um para cada *commit* citado, executamos o CSDiff e o Diff3 para cada cenário, passando como parâmetros para as ferramentas os arquivos em *base*, *left* e *right*, e gravamos os resultados respectivamente como *csmerge* e *d3merge*.

Em seguida, a infraestrutura implementada para a realização da nossa avaliação compara esses resultados gravados e coleta informações sobre ocorrência de conflitos, arquivos conflitantes, etc. Um *conflito* é o texto reportado por uma ferramenta de merge com os marcadores ilustrados na Figura 4. Como uma ferramenta pode reportar mais de um conflito por arquivo, um *cenário com conflito* é um cenário de *merge* em que pelo menos um arquivo contém pelo menos um conflito. Havendo diferença entre os resultados *csmerge* e *d3merge*, os mesmos são comparados com o *rmerge* e mais informações são coletadas, inclusive para análise manual posterior de falsos positivos e falsos negativos, quando necessário.

Um falso positivo, no contexto deste trabalho, ocorre quando um conflito é relatado pela ferramenta mas as mudanças que estão sendo integradas não são excludentes entre si ou não interferem semanticamente uma na outra. Para efeito da nossa avaliação, consideramos a métrica *falso positivo adicionado* (*aFP*, do inglês *added False Positive*). Ao comparar duas ferramentas de *merge* *C* e *D*, contabilizamos um falso positivo adicionado para *C* quando *C* reporta conflito para um dado cenário de *merge*, *D* não reporta conflito no mesmo cenário, e as mudanças que estão sendo integradas pelas ferramentas realmente não interferem uma à outra. É importante utilizar o conceito de falso positivo adicionado porque o conjunto de falsos conflitos de uma ferramenta não é necessariamente subconjunto da outra, e, para os propósitos deste trabalho, não temos interesse em analisar se os cenários em que as duas ferramentas dão o mesmo resultado são falsos positivos ou não.

De forma análoga, um falso negativo ocorre quando um conflito *não* é relatado pela ferramenta de *merge*, mas deveria ter sido reportado pois as mudanças que estão sendo integradas interferem uma na outra. Contabilizamos um *falso negativo adicionado* (*aFN*) para *C* quando *C* não reporta conflito, *D* reporta conflito para o mesmo cenário, e as mudanças que estão sendo integradas interferem uma à outra.

4.2 Amostra

A amostra deste estudo é baseada em nove projetos abertos (*open source*) do Github, que possuem Java como a linguagem majoritária. Os seguintes projetos foram escolhidos: Accumulo, Jackson Databind, Singularity, Spring Boot, Mockito, Spring Framework, Libgdx, Jenkins e FastJson.[5] Para escolher estes projetos, foram observadas as seguintes condições:

- são projetos populares, com base em sua quantidade de estrelas no GitHub. Todos os projetos possuem no mínimo 700 estrelas;
- são projetos que possuem no mínimo 50 colaboradores. Uma grande quantidade de colaboradores pode ser um indicativo de uma maior chance de haver mudanças paralelas que precisam ser integradas através de *merge*;
- são projetos que apresentam *merge commits* em seu histórico de *commits*. Isto é necessário para que seja possível obter os cenários de *merge*.

4.3 Ferramentas e processo

Como a amostra consiste de projetos Java, todo o código analisado nesse experimento é escrito em Java. Dessa forma, tivemos que definir que separadores sintáticos de Java usar como entrada para o CSDiff. Inclusive para entender o efeito causado pelo conjunto de separadores no resultados do CSDiff, optamos por realizar o estudo com dois conjuntos de separadores. Assim, para cada cenário de *merge* da amostra, executamos CSDiff duas vezes.[6] O primeiro conjunto é formado por '`{`', '`}`', '`,`', '`(`', '`)`' e '`;`'. O segundo conjunto é, na verdade, um subconjunto do primeiro contendo apenas os três primeiros separadores listados.

O processo de realização do estudo foi estruturado em três passos: mineração, execução e análise. O processo de mineração tem como objetivo extrair os cenários de *merge* dos nove projetos selecionados. O processo de execução tem como objetivo executar as ferramentas de *merge* (Diff3, CSDiff+ com 6 separadores, CSDiff- com 3 separadores) e salvar os resultados do *merge* para cada cenário. O último

[4] Aparecem como constantes no código, mas poderiam ser facilmente extraídos para parâmetros, por exemplo no caso de uma linguagem ou arquivo que use o marcador de linhas que ilustramos para algum propósito específico.

[5] Mais informações disponibilizadas em nosso Apéndice Online.

[6] Detalhe omitido por simplicidade da explicação na Seção 4.1.

(1) Coleta *commits* de cada projeto por um período de 5 meses;
(2) Seleciona *commits* que são de *merge*;
(3) Para cada *commit* de *merge*, identifica o *base*, *left* e *right*, e, para cada arquivo com versões nesses três *commits*
   (a) Seleciona as triplas em que a versão do arquivo de *left* é diferente de *base*, a de *right* é diferente da de *base*, e a de *left* é diferente da de *right*;
   (b) Executa o Diff3, CSDiff+ e o CSDiff- para as versões dos arquivos em *base*, *left* e *right*, e armazena os três resultados em três arquivos;
(4) Compara os resultados gerados pelos passos anteriores e cria um *csv* com as informações relevantes.

Figure 11: Procedimento da instanciação do *framework*

passo, a análise, tem como objetivo comparar os resultados obtidos pelas ferramentas.

Para automatizar esses três passos, instanciamos um *framework*[7] para análise de repositórios e realização de estudos de integração de código. Precisamos instanciar e estender o *framework* como descrito a seguir. Para a coleta de dados, reusamos e adaptamos componentes existentes de um outro estudo similar. Em relação à execução e análise dos resultados, apesar do código ser semelhante com o do estudo no qual foi baseado o módulo de coleta, foi necessário incluir novos componentes para implementar os requisitos da execução e da análise necessária. Além disso, a implementação do CSDiff precisou ser movida para o projeto do *framework*.

Para coleta dos cenários de *merge*, foi utilizado um período de 5 (cinco) meses de *commits* de cada projeto, começando em 1/10/2021 e indo até 1/03/2021. Como o CSDiff utiliza comandos nativos do Unix, a instanciação do *framework* foi executada localmente numa máquina que opera o sistema operacional Ubuntu (sistema baseado no Linux), versão 20.04.2 LTS, com uma SSD de 256GB (duzentos e cinquenta e seis *gigabytes*), uma memória RAM de 16GB (dezesseis *gigabytes*) e um processador Intel Core i7. O procedimento executado é resumido na Figura 11.

Entre as informações coletadas no último passo do procedimento da Figura 11, temos para cada cenário de *merge* (linha no arquivo) métricas (nas colunas do arquivo) como número de conflitos reportados por cada ferramenta, cenários com conflitos reportados por cada ferramenta, além de falsos positivos e negativos adicionados por cada uma das ferramentas. Para contar conflitos por arquivo, cenário, etc., nossa instanciação do *framework* busca por marcadores de conflito nos textos dos arquivos. Para identificar se as ferramentas deram o mesmo resultado, ou resultado idêntico ao do *merge* do repositório, comparamos textualmente (ignorando espaços em branco) o arquivo de *merge* do repositório (arquivo presente no *commit* de *merge*), e os arquivos resultante da execução de cada ferramenta no cenário de *merge* correspondente.

Para definir se um arquivo apresenta um *aFP* de uma ferramenta C em comparação à ferramenta D, a nossa implementação verifica se o resultado de executar C, nas versões *base*, *left* e *right* desse arquivo, gera conflito, enquanto o resultado de D nos mesmos arquivos não gera conflito e é textualmente igual ao do *merge* do repositório, o que sugere que D produz nesse arquivo o resultado esperado pelos desenvolvedores. Para definir se um arquivo apresenta um possível *aFN* de uma ferramenta C em comparação à ferramenta D, a nossa implementação verifica se o resultado de executar C, nas versões *base*, *left* e *right* desse arquivo, não gera conflito, enquanto Y gera conflito para os mesmos arquivos, e o resultado de C é textualmente diferente do *merge* do repositório. É importante salientar que o resultado de uma ferramenta ser diferente do *merge* do repositório não significa que a ferramenta errou, pois pode ter ocorrido alguma alteração manual no código do *merge* do repositório, por parte do desenvolvedor, após ter realizado a integração com outra ferramenta. Portanto, a nossa implementação somente identifica possíveis *aFN*.

Visando consolidar o número de *aFP* e *aFN* para cada ferramenta em cada cenário, totalizamos os valores observados para os arquivos dos cenários, e realizamos uma análise manual em cada arquivo identificado como contendo um possível *aFN*, comparando o resultado da ferramenta com o *merge* do repositório, observando se as diferenças entre eles eram alterações manuais realizadas pelo desenvolvedor, e se havia de fato interferência entre as mudanças. Caso as diferenças fossem alterações do próprio desenvolvedor feitas durante o *merge*, a ferramenta teria integrado corretamente, o sque ignifica que aquele arquivo não contem um *aFN* da ferramenta, e sim um *aFP* da outra ferramenta. A maior parte das avaliações manuais foram realizadas por um desenvolvedor com mais de 2 anos de experiência, que analisou casos simples como comentários removidos não conflitantes, e adições no resultado final mas que não estavam em nenhuma *branch*. Os outros casos foram adicionalmente revisados por duas pessoas mais experientes, uma com mais de 30 anos de experiência, e outra com mais de 8 anos. No total, 42 arquivos de 38 cenários de *merge* foram analisados manualmente.

4.4 Resultados

Com o objetivo de avaliar o CSDiff e entender como a escolha dos separadores impacta nos resultados, comparamos o CSDiff ao Diff3, utilizando os dois conjuntos de separadores[8] apresentados na Seção 4.1. Para a nossa amostra, foram coletados 3721 cenários de *merge* dos cinco meses de histórico analisado dos nove projetos considerados. Destes, apenas 1020 cenários foram utilizados para a comparação, pois possuíam pelo menos um arquivo em comum modificado pelos dois commits *left* e *right*. Os cenários descartados não interessam para a nossa análise porque as três ferramentas geram os mesmos resultados quando arquivos comuns não são modificados.

PP1: O CSDiff reduz a quantidade de conflitos reportados em comparação ao Diff3?

Para responder a PP1, primeiro observamos o total de conflitos detectados, por cada ferramenta, na nossa amostra, que é apresentado na primeira linha da Tabela 1. O CSDiff aumenta em aproximadamente 105% o número de conflitos para o conjunto maior de separadores (CSDiff+), e em aproximadamente 35% para o conjunto menor de separadores (CSDiff-).

Esses aumentos ocorrem por uma questão puramente de granulosidade dos conflitos; uma dada mudança conflitante que leva o Diff3 a reportar um único conflito, pode fazer com que o CSDiff

[7] Link da Mining Framework.

[8] Destes conjuntos, é ignorado o caráter de quebra de linha, pois o CSDiff executa o Diff3 em seu algoritmo, logo a quebra de linha já está inclusa através do Diff3.

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

reporte vários conflitos para a mesma mudança. Ou seja, não é que o CSDiff sinalize, de forma significativa, conflitos em situações em que o Diff3 não sinaliza; é que, para a mesma situação, o CSDiff sinaliza vários conflitos menores ao invés de um único conflito maior.[9] Isso também foi observado para ferramentas estruturadas em estudos anteriores [11].

Uma abordagem híbrida simples como executar o Diff3 toda vez que o CSDiff resultar em conflito, resolveria este problema, mantendo os conflitos de maior granulosidade do Diff3— e assim reduzindo o número de conflitos— ao mesmo tempo em que reduz a quantidade de falsos positivos e cenários com conflitos, como explicado mais adiante. Outra indicação de que o maior número de conflitos reportados pelo CSDiff não é um problema maior, no sentido de que aumenta o esforço em resolver conflitos e impacta negativamente na produtividade da equipe [5, 7], é dada pela redução apresentada no número de arquivos com conflitos (segunda e terceira linhas da Tabela 1). A redução de arquivos com conflitos é percentualmente bem pequena, mas isso se deve a existência de 8 cenários, dentre os 1020 cenários coletados, que são *outliers*, pois possuem juntos 7179 arquivos com conflitos para ambas as ferramentas. Desconsiderando esses cenários, mostramos na terceira linha da tabela uma redução de aproximadamente 4.8% de arquivos com conflitos para o CSDiff+ e aproximadamente 1% para o CSDiff-. Isso implica que desenvolvedores que usem o CSDiff, pelo menos na amostra considerada, teriam que resolver conflitos em menos arquivos e, com a solução híbrida sugerida, resolveriam menos conflitos também.

PP2: O CSDiff reduz a quantidade de cenários com conflitos reportados em comparação ao Diff3?

Como pode ser observado na quarta linha da Tabela 1, a resposta da PP2 é positiva. Para ambos os conjuntos de separadores, é observada na nossa amostra uma redução no número de cenários com conflitos quando se usa o CSDiff. Isso sugere que usuários do CSDiff precisarão resolver conflitos em menos situações de integração de código do que usuários do Diff3, trazendo benefícios para o processo de integração de código sem demandar os custos extras das ferramentas estruturadas e semiestruturadas. A redução é de aproximadamente 5% para o conjunto maior de separadores (CSDiff+), e de 2% para o conjunto menor (CSDiff-). Estes resultados indicam que a quantidade de separadores escolhidos impacta na redução de cenários com conflitos, e a execução com mais separadores de Java poderia aumentar a dimensão da redução obtida pelo CSDiff.

[9]Mais detalhes podem ser encontrados em nosso Apêndice Online.

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

Apesar da redução de cenários com conflitos favorável ao CSDiff, ela não é tão significativa. Isso ocorre pelo fato de que ambas possuem muitos cenários com conflitos, já que a existência de um único arquivo com conflito já marca o cenário como conflitante. Ainda que houvesse uma divergência maior entre os cenários, não seria suficiente para determinar que o CSDiff reduz a quantidade de cenários com conflitos. A ferramenta proposta só levará vantagem quando houverem mudanças a serem integradas em uma mesma linha (ou linhas consecutivas) e haja um separador (como "{" ou "}") entre as mudanças. Projetos em que situações como essa não ocorram ou sejam raras, as ferramentas devem essencialmente apresentar o mesmo comportamento.

PP3: O CSDiff reduz a quantidade de falsos conflitos, e cenários com falsos conflitos, reportados (falsos positivos) em comparação ao Diff3?

Para responder a PP3, analisamos a quantidade de *aFP* de cada ferramenta. Como comentado na Seção 4, as ferramentas comparadas podem possuir falsos positivos em comum, mas o que interessa para a nossa análise é o conjunto de falsos positivos reportados (ou "adicionados", daí a sigla *aFP*) por uma ferramenta mas não pela outra. Isso é o que ajuda a determinar a possível desvantagem de uma ferramenta em relação à outra, que é o que queremos entender com esse estudo. Assim, observamos na terceira linha da Tabela 2 uma clara desvantagem do Diff3 em relação ao CSDiff+ (que usa o conjunto maior de separadores) em relação ao número de cenários com *aFP*. Isso mostra que em 30 dos 34 (primeira linha da tabela) cenários em que somente o Diff3 relatou conflito, o CSDiff+ foi capaz de integrar as mudanças corretamente, enquanto que o Diff3 sinalizou um falso conflito. Nessa amostra, quando as ferramentas dão resultados diferentes (conflito versus não conflito) em um determinado cenário, quase sempre o CSDiff+ acerta. De fato, a partir do resultado apresentado na Seção 4.4 de que houve uma redução de aproximadamente 5% na quantidade de cenários com conflitos, pode ser observado que em aproximadamente 88% dos cenários reduzidos, o CSDiff realiza corretamente o *merge*.

De forma similar, fizemos uma análise de *aFP* por arquivo, ao invés de cenário, por existir a possibilidade de um arquivo conter um *aFP* de uma ferramenta mas o cenário como um todo ter também verdadeiros positivos, o que faz com que o cenário como um todo não seja contabilizado como um *aFP*. Analisando a segunda e a quarta linha da Tabela 2, observamos um padrão similar ao observado para cenários, com clara vantagem para o CSDiff+ também considerando *aFP* por arquivo. O Diff3 apresenta 63 *aFPs* em uma análise por arquivo, mostrando que em aproximadamente 81% dos arquivos onde somente o Diff3 resultou em conflito, o CSDiff corretamente gerou um resultado sem conflitos. Novamente, quando as

Table 3: Resultados do CSDiff- em comparação ao Diff3

#	Diff3	CSDiff-
Cenários com conflitos exclusivos	14	5
Arquivos com conflitos exclusivos	21	5
aFP Cenários	14	2
aFP Arquivos	20	5
aFN Cenários	0	0
aFN Arquivos	0	1

ferramentas dão resultados diferentes (conflito versus não conflito) em um determinado arquivo, quase sempre o CSDiff+ acerta.

Resultados no mesmo sentido de apontar vantagem para o CSDiff em termos de *aFP* foram observados quando comparando o Diff3 com o CSDiff-, que considera um conjunto menor de separadores, subconjunto do considerado pelo CSDiff+. No entanto, a intensidade da vantagem, em termos absolutos, é reduzida quando considera-se menos separadores, como mostrado na Tabela 3. Para o conjunto menor de separadores, 14 cenários de *merge* (terceira linha) apresentaram falsos positivos adicionados pelo Diff3, o que é menos da metade observada quando comparando o Diff3 com o CSDIff+. Por outro lado, em 14 dos 14 cenários (primeira e terceira linhas) nos quais somente o Diff3 relatou conflito, o CSDiff-foi capaz de integrar as mudanças corretamente, enquanto o Diff3 sinalizou conflito. Logo, em 100% dos cenários onde só o Diff3 retornou conflito, ele errou. Na métrica por arquivos, em 20 dos 21 arquivos (segunda e quarta linhas) onde somente o Diff3 relatou conflito, o CSDiff- acertou, representando uma taxa de acerto de 95%.

Apesar de o Diff3 ter apresentado mais cenários onde só ele reportou conflitos, a grande maioria sendo falsos conflitos, o CSDiff também relatou alguns poucos falsos conflitos em relação ao Diff3, como pode ser observado na terceira e quarta linhas das Tabelas 2 e 3. A ocorrência desses falsos conflitos se deve ao fato de que o CSDiff isola os separadores em novas linhas, o que faz com que, por algumas vezes, ocorra um alinhamento diferente do esperado, "confundindo" o Diff3 na busca por áreas de texto que foram modificadas por *left* e *right*. Por exemplo, num cenário do projeto Jackson Databind, um desenvolvedor (*left*) modificou um arquivo adicionando mais asteriscos ao final da primeira linha de um comentário de 3 linhas que começa com "/****". Em outra parte do arquivo, um outro desenvolvedor (*right*) adicionou um novo comentário, parecido com o modificado por *left*, só que com a mesma quantidade de asteriscos de *base* e um texto que tem como prefixo o texto da segunda linha do comentário alterado por *left*. Como o sufixo desse texto começa com o separador ',', o CSDiff quebrou o prefixo e o sufixo em duas linhas, e alinhou o comentário alterado por *left* com o comentário adicionado por *right* (já que, após a quebra, os dois passam a ter o mesmo texto na segunda linha), sinalizando conflito, apesar de *left* e *right* terem trabalhado em comentários diferentes.[10] Uma versão do CSDiff que não considera separadores dentro de comentários teria evitado esse falso conflito, como discutido na Seção 4.5.

Os resultados apresentados nesta seção sugerem que o CSDiff tem potencial para evitar parte dos falsos conflitos sinalizados pelo Diff3, e assim impactar positivamente na produtividade do processo de integração de código. Além disso, temos evidência de que esse potencial é influenciado pela quantidade de separadores fornecidos para a ferramenta, com um maior número de separadores implicando em maior potencial de redução de falsos positivos. Apesar de essa redução parecer ser menos expressiva do que a redução obtida por ferramentas semiestruturadas para Java [2, 9, 10], ela é complementar na maioria das situações que observamos. De fato, os falsos positivos evitados por ferramentas semiestruturadas são os chamados conflitos de ordenamento, que normalmente ocorrem entre declarações de método e atributos em uma classe, mas nunca nos corpos dos métodos. Já o CSDiff tem o potencial para evitar falsos positivos inclusive dentro dos corpos dos métodos, já que lida com mudanças ocorrendo na mesma linha ou em linhas consecutivas e que são separadas por separadores sintáticos, não importando que partes do programa estejam associadas à essas linhas.

PP4: Em comparação ao Diff3, o CSDiff aumenta o número de integrações de mudanças que interferem uma na outra sem reportar conflitos (falsos negativos), e de cenários com tais integraçõe?

Para adoção na prática, não basta que uma ferramenta de *merge* apresente uma redução no número de falsos positivos, e cenários e arquivos com conflitos. Se essa redução for obtida em detrimento da corretude do processo de integração, elimina-se o interesse na ferramenta. Por isso, analisamos agora a taxa de integrações erradas de cada ferramenta considerada no nosso estudo e, assim, respondemos PP4 focando na quantidade de *aFN* de cada ferramenta.

Para o conjunto maior de separadores, a quinta linha da Tabela 2 revela uma desvantagem do CSDiff+ considerando a métrica *aFN*. Apesar de, em termos absolutos, a desvantagem ser pequena em relação ao Diff3, e principalmente bem inferior ao ganho observado em termos de *aFP*, precisa ser considerada. Em resumo, em 4 dos 34 cenários (primeira linha) onde somente o Diff3 reportou conflito, o CSDiff+ integrou erroneamente pelo menos um arquivo do cenário, representando aproximadamente uma taxa de erro de 12% (contrastando com os 88% de acerto e redução de falso positivo discutido na Seção~4.4). Em relação à métrica por arquivos, os resultados apontam na mesma direção mas com uma intensidade maior: em 15 (sexta linha) dos 78 arquivos onde somente o Diff3 deu conflito, o CSDiff errou, representando uma taxa de aproximadamente 19%.

Como comentado na Subseção 4.4, como o CSDiff cria várias linhas novas, uma para cada separador, em algumas situações temos alinhamentos diferentes ao executar o Diff3 para esses arquivos. Esses alinhamentos problemáticos ocorrem porque a ferramenta encontra uma maior correspondência entre arquivos usando as novas linhas, menores, criadas pelo algoritmo. Esse problema, além de gerar falsos conflitos, faz com que o CSDiff, em alguns casos, erre em não gerar conflito num processo de *merge*.[11]

Para entender como o falso alinhamento leva a um *aFN*, considere um arquivo *base* que tem a linha "`a().b(c).d();`", com chamadas encadeadas de métodos. Suponha que a desenvolvedora *left* modificou o argumento do método `b`, resultando em "`a().b(e).d();`". Enquanto isso, a desenvolvedora *right* trocou a chamada ao método `b` por uma chamada à `g`, da seguinte forma: "`a().g(h(c)).d();`". Ao integrar essas versões, o CSDiff, utilizando os parênteses como separadores, acaba gerando um alinhamento entre a parte "*(c)*" de

[10] Mais detalhes podem ser encontrados em nosso Apêndice Online.

[11] Mais detalhes podem ser encontrados em nosso Apêndice Online.

base com a de *right*. Por adicionar uma nova linha antes e depois de cada separador, o CSDiff entende que essas linhas formam o maior alinhamento e, por causa disso, a versão de *left* teria somente modificado o parâmetro de `c` pra `e`, aceitando ambas as modificações sem conflito e resultando em "`a().g(h(e)).d();`". No entanto, havendo dependências semânticas entre as modificações, o CSDiff deixaria escapar um conflito.

No caso do conjunto menor de separadores, a perda do CSDiff em termos de *aFP* é bem menor. Em termos de cenários, o CSDiff- acertou em todos os cenários em que ele não retornou conflito, não possuindo *aFN* em relação ao Diff3 (quinta linha da Tabela 3). Em termos de arquivos, o CSDiff- apresentou apenas um *aFN*, mesmo assim um erro característico do Diff3, não do CSDiff. Porém, como o Diff3 reportou conflito em outra parte do mesmo arquivo, foi contabilizado, conservadoramente, como um *aFN* de arquivo para o CSDiff em relação ao Diff3.

Os resultados apresentados nesta seção sugerem que o CSDiff pode impactar negativamente a corretude do processo de integração de código e a qualidade dos sistemas, mas de forma bem menos intensa do que o impacto positivo que ele pode trazer em termos de redução de falsos positivos e potencial aumento da produtividade da equipe. Além disso, alguns desses casos de *aFN* poderiam ter sido evitados solucionado o problema do alinhamento da versão atual da ferramenta. Por fim, a quantidade de cenários com falsos negativos adicionados, no caso do conjunto maior de separadores, representa aproximadamente 0.4% do número total de cenários. Por estar tentando simular a abordagem estruturada, é interessante perceber que essa taxa se mostra bem menor do que a produzida pela ferramenta estruturada observada por Cavalcanti et. al [11]. Percebe-se, também, que a quantidade de separadores influencia no aumento ou redução de integrações erradas, onde uma menor quantidade de separadores implica em redução de erros e uma maior quantidade implica num aumento de erros.

4.5 Discussão

Apesar de vantagens importantes apresentadas pelo CSDiff, nosso protótipo de ferramenta de *merge* não estruturado com separadores, acreditamos que há espaço para explorar ganhos mais significativos com este tipo de ferramenta e, principalmente, com ferramentas híbridas, que combinem o CSDiff com outras ferramentas não estruturadas e semiestruturadas. De qualquer forma, acreditamos que que esse estudo é relevante por propor e analisar de forma inicial um novo tipo de ferramenta de *merge* textual, entendendo o potencial dessa estratégia isoladamente, antes de tentar soluções mais elaboradas e que possam combinar efeitos de abordagens diferentes.

Como discutido na Seção 4.4, o CSDiff, para ambos os conjuntos de separadores, gerou mais conflitos do que o Diff3. Como discutido anteriormente, isso ocorre porque a ferramenta divide os trechos de código em mais linhas (utilizando separadores), o que torna o conflito mais granular. Uma possível solução seria simplesmente retornar o resultado do Diff3 para os arquivos originais, caso o CSDiff retornasse conflito. Isto é, poderia ser implementada uma nova ferramenta de *merge* que primeiro executa o CSDiff apresentado aqui. Caso ele não sinalize conflito, a ferramenta simplesmente retornaria o resultado do CSDiff. Caso contrário, a ferramenta executaria o Diff3 e retornaria o seu resultado. Assim, essa nova ferramenta manteria as vantagens do CSDiff com relação à redução de falsos positivos ao mesmo tempo em que mantém os conflitos de maior granulosidade do Diff3, reduzindo também o número de conflitos reportado.

Um outro aspecto que pode ser melhorado é a redução da taxa de cenários de *merge* sem conflito, como discutido na Seção 4.4. Acreditamos que o ganho apresentado não é maior porque o CSDiff é essencialmente uma ferramenta não estruturada. Apesar de evitar conflitos que também são evitados por ferramentas estruturadas, o CSDiff não consegue evitar alguns tipos de conflito, como os de ordenamento, que são resolvidos pelas ferramentas estruturadas e semiestruturadas [1, 2, 9–11]. Apesar de as ferramentas estruturadas, a um custo maior, resolverem conflitos similares ao CSDiff, as ferramentas semiestruturadas não conseguem. Assim, uma ideia seria explorar uma solução híbrida em que uma ferramenta de *merge* semiestruturado chame o CSDiff, ao invés de utilizar o *merge* não estruturado puramente textual, nas folhas da árvore sintática (nós que contém blocos de código representados como *strings*). Essa integração, possivelmente, contribuiria em uma maior redução na quantidade de falsos conflitos, por combinar as vantagens dos dois tipos de ferramenta, a um custo menor do que com ferramentas estruturadas.

As ideias apresentadas nesta seção são possibilidades baseadas nos resultados analisados. É importante salientar que elas necessitam de estudos futuros para avaliá-las.

4.6 Ameaças à validade

Os resultados e avaliações naturalmente abrem espaço para algumas ameaças à validade. Primeiro, por existir a possibilidade de o desenvolvedor ter modificado o código durante o processo de *merge*, a ferramenta utilizada para análise não é capaz de identificar com certeza se um cenário ou arquivo é um falso negativo adicionado, por isso, foi realizada uma análise manual para os casos de possíveis *aFNs*. Por esta análise ter sido manual, existe a possibilidade de ter ocorrido um julgamento errado em avaliar os casos de possíveis falsos negativos, apesar de termos mitigado os riscos seguindo um protocolo e envolvendo mais de um autor no processo, um deles com significativa experiência nesse processo.

Para o estudo, foram utilizados projetos abertos do GitHub. Esses projetos, podem ter sofrido alterações no seu histórico de *commits*, tais como utilizar operações como *rebase* ou *cherry-pick*, que podem ter influenciado em uma perda cenários de *merge*. Além disso, são projetos diferentes que, possivelmente, adotam estilos diferentes de código, tornando possível haver mais ou menos mudanças entre separadores, a depender do projeto, o que impacta na performance do CSDiff. Por fim, o estudo foca na linguagem de programação Java. Por isso, não é possível afirmar que o CSDiff se comportará da mesma maneira para outras linguagens ou mesmo para outros conjuntos de separadores.

5 TRABALHOS RELACIONADOS

Várias ferramentas foram criadas com o objetivo de aprimorar o processo de *merge* reduzindo e o esforço da integração e aprimorando a corretude do processo. Westfechtel [23] e Buffenbarger [8] foram pioneiros quando se trata de propor soluções que utilizam a estrutura do arquivo para realizar o *merge*. Posteriormente, outros

pesquisadores implementaram soluções que se baseavam na estrutura de linguagens de programação específicas, como por exemplo Java [3] e C++ [13].

Visando reduzir o custo de criação de ferramentas estruturadas, a abordagem semiestruturada foi proposta. Por exemplo, Apel et al. [2] propuseram e avaliaram a ferramenta semiestruturada *FSTMerge*. Alguns estudos [2, 9, 10], utilizando o *FSTMerge*, mostraram que a abordagem semiestruturada reduz o número de falsos conflitos gerados pela não estruturada, porém, essa redução não ocorreu para todos os projetos e cenários analisados. Descobertas semelhantes, mas em menor grau, são observadas por Trindade et al. [22] ao investigarem o *merge* semiestruturado em projetos Javascript.

Apel et al. [1] também propuseram uma ferramenta estruturada, chamada *JDime*, que é capaz de ajustar o processo de *merge* em tempo real, alternando entre o estruturado e não estruturado a depender da ocorrência de conflitos. Seguindo a mesma linha, Zhu et al. [25] propuseram a *AutoMerge*, uma ferramenta que baseada no *JDime*. Eles conseguiram reduzir o número de falsos conflitos, comparado ao *JDime* original. Além disso, o estudo realizado por Cavalcanti et al. [11] comparou a abordagem semiestruturada com a estruturada e constatou que, apesar da semiestruturada tender a gerar mais falsos conflitos, a estruturada gera uma quantidade maior de falsos negativos.

Por fim, ferramentas baseadas em estratégias semânticas também foram propostas [4, 14, 15, 24], mas por muito tempo têm-se demonstrado impraticáveis. Num esforço recente para viabilizar ferramentas semânticas, Sousa et. al [21] propuseram *SafeMerge*, uma ferramenta semântica que verifica se o *merge* de programas não introduz novos comportamentos indesejados. Eles descobriram que a ferramenta proposta pode identificar problemas comportamentais em integrações problemáticas que são gerado por ferramentas não estruturadas. Ainda, para reduzir conflitos, Rocha et al. [20] propuseram uma ferramenta que, no contexto de BDD (desenvolvimento orientado por comportamento), analisa estaticamente testes para inferir quais arquivos serão modificados pelos desenvolvedores, evitando a execução paralela de tarefas que potencialmente levarão a conflitos de código.

6 CONCLUSÃO

Propomos neste artigo um novo tipo de ferramenta textual de *merge*, que explora parte da estrutura sintática da linguagem, mas sem a necessidade de criar e manipular árvores. Além disso, comparamos a acurácia dessa ferramenta (CSDiff) em relação ao *merge* não estruturado textual tradicional. Os resultados obtidos mostram que o CSDiff reduz o número de *falsos positivos* e *cenários com conflitos* em comparação ao *diff3*. Essa redução sugere que o CSDiff pode trazer benefícios para o processo de integração de código sem demandar os custos extras das ferramentas estruturadas e semiestruturadas. Além disso, na grande maioria das situações em que o CSDiff diverge do Diff3, o CSDiff dá o resultado correto enquanto o Diff3 erra sinalizando falsos conflitos. Já com relação a *falsos negativos*, o CSDiff apresenta um aumento quando comparado com o *diff3*, mas proporcionalmente bem menor que a redução em falsos positivos.

REFERENCES

[1] Sven Apel, Olaf Leßenich, and Christian Lengauer. 2012. Structured merge with auto-tuning: balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. ACM Press. https://doi.org/10.1145/2351676.2351694

[2] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured merge. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*. ACM Press. https://doi.org/10.1145/2025113.2025141

[3] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2006. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering* 14, 1 (Dec. 2006), 3–36. https://doi.org/10.1007/s10515-006-0002-0

[4] David Binkley, Susan Horwitz, and Thomas Reps. 1995. Program Integration for Languages with Procedure Calls. *ACM Transactions on Software Engineering and Methodology* (1995).

[5] Christian Bird and Thomas Zimmermann. 2012. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*. ACM Press. https://doi.org/10.1145/2393596.2393648

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

[7] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*. ACM Press. https://doi.org/10.1145/2025113.2025139

[8] Jim Buffenbarger. 1993. Syntactic software merging. In *Software Configuration Management*. Springer, 153–172.

[9] Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. 2015. Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. https://doi.org/10.1109/esem.2015.7321191

[10] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 1–27. https://doi.org/10.1145/3133883

[11] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. 2019. The Impact of Structure on Software Merging: Semistructured Versus Structured Merge. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. https://doi.org/10.1109/ase.2019.00097

[12] W Keith Edwards. 1997. Flexible conflict detection and management in collaborative applications. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*. 139–148.

[13] Judith E. Grass. 1992. Cdiff: A Syntax Directed Differencer for C++ Programs. In *Proceedings of the USENIX C++ Conference*. USENIX Association.

[14] Susan Horwitz, Jan Prins, and Thomas Reps. 1989. Integrating Noninterfering Versions of Programs. *ACM Transactions on Programming Languages and Systems* (1989).

[15] Jackson and Ladd. 1994. Semantic Diff: a tool for summarizing the effects of modifications. In *Proceedings 1994 International Conference on Software Maintenance (ICSM'94)*. IEEE.

[16] Sanjeev Khanna, Keshav Kunal, and Benjamin C Pierce. 2007. A formal investigation of diff3. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 485–496.

[17] Ali Koc and Abdullah Uz Tansel. 2011. A survey of version control systems. *ICEME 2011* (2011).

[18] Tancred Lindholm. 2004. A three-way merge for XML documents. In *Proceedings of the 2004 ACM symposium on Document engineering - DocEng '04*. ACM Press. https://doi.org/10.1145/1030397.1030399

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

[20] Thaís Rocha, Paulo Borba, and João Santos. 2019. Using acceptance tests to predict files changed by programming tasks. *Journal of Systems and Software* (2019), 176–195.

[21] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. 2018. Verified Three-way Program Merge. *Proceedings of the ACM on Programming Languages (OOPSLA)* (2018).

[22] Alberto Trindade, Paulo Borba, Guilherme Cavalcanti, and Sergio Soares. 2019. Semistructured Merge in JavaScript Systems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. ACM.

[23] Bernhard Westfechtel. 1991. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd international workshop on Software configuration management -*. ACM Press. https://doi.org/10.1145/111062.111071

[24] Wuu Yang, Susan Horwitz, and Thomas Reps. 1990. A Program Integration Algorithm That Accommodates Semantics-preserving Transformations. *SIGSOFT Software Engineering Notes* (1990).

[25] Fengmin Zhu, Fei He, and Qianshan Yu. 2019. Enhancing precision of structured merge by proper tree matching. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 286–287.