Aperfeiçoamento na detecção e resolução de conflitos utilizando merge semiestruturado

Giovanni Barros (gaabs@cin.ufpe.br)

Universidade Federal de Pernambuco

Centro de Informática

Resumo

No desenvolvimento de software, é comum que desenvolvedores trabalhem paralelamente fazendo alterações na mesma base de código. Quando versões diferentes são combinadas, podem ocorrer conflitos de integração. Soluções de merge semiestruturado e estruturado fazem uso da estrutura sintática e semântica dos elementos envolvidos, sendo mais precisas do que o merge não estruturado. No entanto, mesmo as alternativas semiestruturadas trazendo vários avanços, ainda há bastante espaço para melhorias, principalmente na detecção de conflitos. Um dos casos mais frequentes de conflitos é o de renaming, quando dois desenvolvedores modificaram o mesmo método e pelo menos um modificou a respectiva assinatura, sendo o foco do estudo. Este trabalho, assim, teve como objetivo estender uma solução semiestruturada, provendo novas estratégias para resolução de conflitos de renaming. Posteriormente, apresenta o resultado dos experimentos e análises manuais a partir de cenários de merge de repositórios do GitHub na linguagem Java.

Palavras-chave: Desenvolvimento Colaborativo de Software, Conflitos de Integração, Engenharia de Software, Estudos Empíricos

1. Introdução

No desenvolvimento de software, é comum que membros diferentes trabalhem simultaneamente em atividades de um mesmo projeto. Como ambas tarefas podem alterar ou fazer uso de uma parte comum do código, é possível que surjam efeitos indesejados decorrentes da combinação das mudanças, sendo estes denominados conflitos.

13 de Dezembro de 2018

A resolução manual de conflitos, no entanto, pode ser uma tarefa cansativa e tediosa, sendo um empecilho à produtividade, além de ser passível de erros. É possível, ainda, que os conflitos não sejam detectados e comprometam a corretude do sistema [1]. Inclusive, estudos mostram que mais de 23% dos cenários de merge apresentam conflitos [2], sendo mais uma evidencia da necessidade de técnicas de detecção e resolução automática de conflitos.

A comparação não estruturada é a forma mais comumente utilizada no auxílio à resolução de conflitos de integração. É uma técnica relativamente simples, tendo assim suas falhas. Por apenas detectar conflitos em áreas de edição comum, perde de reportar possíveis problemas relacionados quando ocorrem em áreas distintas. Além disso, parte dos conflitos reportados são bem simples e poderiam ser resolvidos automaticamente.

Comparações semiestruturadas e estruturadas são alternativas mais robustas, que fazem uso da estrutura sintática do código para detectar e solucionar conflitos. O problema da estruturada é que exige um grande tempo de execução [3], sendo um fator que impede a adoção prática. A semiestruturada, por sua vez, é um meio termo, que busca unir o melhor dos dois. Utiliza a estrutura sintática, mas em alguns elementos, como métodos, apica o não estruturado, tendo um tempo de execução razoável [1].

Dentre as alternativas semiestruturadas, destaca-se o s3m [1]. Trata-se de uma solução desenvolvida no meio acadêmico, com foco na linguagem Java, e será a ferramenta utilizada como base para o presente trabalho.

No entanto, mesmo com as alternativas semiestruturadas trazendo vários avanços em relação ao não estruturado, ainda há bastante espaço para melhorias. É possível que seja reportado um conflito que não deveria ter sido reportado, com match entre elementos que não correspondem realmente; ou que um conflito que não foi reportado quando deveria, potencialmente trazendo um erro de integração.

Dentre os vários possíveis casos de conflitos, um de destaque é o de renaming. Estudos mostraram que mais de 40% das refatorações envolvem renaming [4]. Dentre os conflitos de merge, por sua vez, cerca de 25% dos casos contém renaming [5].

O s3m previamente citado, no entanto, possui um tratamento simplificado para lidar com renaming. Sua estratégia original busca basicamente não ser pior do que o não estruturado, de forma a não introduzir conflitos adicionais. Tem, por isso, suas vantagens, sendo um facilitador para sua adoção, mas termina perdendo a chance de fazer uso de informações que o não estruturado não possui, uma vez que reporta conflito apenas quando o não estruturado

também o faz.

Devido à essa possibilidade de melhora no tratamento de renaming, o presente trabalho objetiva desenvolver e analisar formas alternativas para tratar o problema em questão. Foram desenvolvidos dois algoritmos, um que sempre mantém os dois nós envolvidos no conflito, e outro que busca uní-los, independentemente do resultado do não estruturado.

Para análise dos resultados, foram utilizados 1731 cenários de merge na linguagem Java, de modo a comparar o número de conflitos de renaming reportados pelo algoritmo original e pelas soluções desenvolvidas. Além disso, parte dos casos foram selecionados para uma análise manual e qualitativa.

Observou-se que a solução de manter ambos os nós, apesar de reduzir o número de conflitos reportados, teve um aumento significativo de falsos negativos, ou seja, conflitos não acusados e que acarretam falha de integração.

A solução de unir os nós baseados na similaridade, por sua vez, teve um aumento na detecção de conflitos verdadeiros em relação à estratégia original, por não depender do não estruturado. Dessa forma, mais conflitos que deveriam ser reportados passaram a ser devidamente detectados.

A estrutura do trabalho divide-se em seções. A Seção 2 disserta sobre a motivação, descrevendo em mais detalhes as técnicas de merge e os problemas relacionados. A Seção 3 explica a solução, descrevendo as técnicas desenvolvidas. A Seção 4 explica como foi realizado o estudo e seu processo. A Seção 5 apresenta os resultados do estudo com sua análise comparativa. A Seção 6 descreve alguns dos mais importantes trabalhos relacionados. Por fim, a Seção 7 apresenta a conclusão do trabalho.

2. Motivação

Atualmente, é bastante disseminado o uso de sistemas de versionamento de código, como o Git. Fazendo uso de sistemas como esse, dois desenvolvedores podem trabalhar paralelamente em versões diferentes do código, cada um fazendo suas modificações. Quando versões diferentes são combinadas, é necessário verificar se ocorreram conflitos e resolvê-los corretamente.

A forma mais comum de combinação de versões de código é utilizando o merge não estruturado. Nele, os arquivos que foram modificados por ambas as partes são comparados textualmente, com base no algoritmo *diff3* [6], de forma a detectar conflitos se ambas versões alteraram a mesma parte do arquivo. Essa forma de verificação, no entanto, é bastante simples e propensa

a erros, uma vez que os conflitos não surgem apenas nas linhas que ambos modificaram.

O merge estruturado, por sua vez, faz uso dos elementos da linguagem em questão. Nele, os elementos do programa são representados como nós de uma árvore, sendo esse tipo de informação utilizada para resolver automaticamente os conflitos a partir de comparações estruturais. É uma forma bem mais precisa de detectar os conflitos, porém, é menos eficiente em relação ao tempo de execução [3].

Outra técnica existente é o merge semiestruturado [7], que se trata de uma mistura das duas abordagens anteriores: faz-se uso da comparação estruturada quando possível, mas também utiliza merge não estruturado, de forma a atingir um bom desempenho na detecção de conflitos e com custo computacional razoável. Como o estruturado, constrói uma árvore sintática do programa, mas certos nós, como métodos e construtores, são folhas com conteúdo textual. O merge das folhas, por exemplo, faz uso do merge não estruturado.

O s3m [1] é uma solução semi-estruturada para a linguagem java, desenvolvida em cima do FSTMerge [7]. É uma ferramenta que traz várias melhorias em relação à original, trazendo uma redução significativa no número de conflitos erroneamente reportados, e um número relativamente pequeno de falhas adicionais.

O caso de renaming, para soluções semiestruturadas e estruturadas, não é tão simples [7]. Isso porque, quando um método é renomeado, há, simultaneamente, a remoção de um nó, que que possui o conteúdo original, e a adição de um novo nó, com as modificações. Assim, não é trivial detectar que houve um renaming, uma vez que é necessário fazer uma conexão entre o nó removido e o adicionado, seja através de heurísticas, medidas de similaridade ou análise de dependências.

Para o caso de renaming, o s3m, por exemplo, ainda possui espaços para melhorias. O algoritmo atual da ferramenta tem o objetivo de não ser pior do que o não estruturado, idealmente reportando conflito sempre que o não estruturado reportar. Caso contrário, ambos os nós envolvidos no conflito são mantidos. O maior problema disso é que perde-se a possibilidade de utilizar o conhecimento sintático e estrutural para fazer escolhas melhores. Um método alterado por ambas as partes, se for movido para áreas distintas, deixa de ser reconhecido como conflito se o não estruturado não detectar, como pode ser visto na Figura 1. A parte em vermelho representa o método renomeado por ambas as partes, sendo um nó removido em comum. Em

verde e azul estão as novas versões do método, renomeadas e movidas para locais distintos por cada um dos desenvolvedores.

```
void safeShutdown() {
  shutdown(Mode.SAFE);
}
```

[...]

```
void stop() {
  shutdown(Mode.SAFE);
}
```

[...]

```
void safeStop() {
  shutdown(Mode.SAFE);
}
```

Fig. 1: Renaming mútuo com mudança de posição

Uma outra questão é a necessidade de utilização da similaridade. Mesmo que o não estruturado reporte conflito, é possível que o método seja bastante dissimilar entre as duas versões, não sendo detectado o conflito pelo algoritmo. Também é possível que ocorra um match errado, devido ao nó mais similar não ser o que corresponde ao método comumente alterado, mesmo tendo o não estruturado reportado conflito.

Além disso, foram detectadas algumas limitações na implementação atual. Para o caso de renaming mútuo, ou seja, quando ambas as partes renomeiam o método, o programa exige que os corpos sejam iguais à base. Assim, se houve qualquer alteração no corpo, o conflito deixa de ser detectado.

Também há um problema na identificação dos conflitos reportados pelo não estruturado. Como o s3m se baseia no não estruturado, é necessario identificar se o nó que está envolvido em um possível conflito pelo semiestruturado também está presente em um conflito do não estruturado. A forma adotada foi exigir que a assinatura inteira esteja presente o conflito, o que nem sempre é o caso, uma vez que apenas parte dos parâmetros podem estar no conflito, por exemplo. Há também o caso do conflito ser apenas no corpo, quando apenas uma das partes renomeia e há alteração no corpo, mas em

áreas disjuntas. Como não é reportado o conflito na assinatura, o algoritmo também não detecta o conflito. Um caso como este pode ser visto na Figura 2. Em amarelo, o método original; em verde, a assinatura foi modificada por uma das partes; em azul, o corpo do método foi modificado pela outra partes.

```
<<<<<
void processStorage(Context ctx) {
  setup();
  ctx.store();
}
|||||
void process(Context ctx) {
  setup();
  ctx.store();
}
=====
void process(Context ctx){
  setup();
  if (enabled) {
    ctx.store();
  }
}
>>>>>
```

Fig. 2: Renaming mútuo com mudança de posição

Quando o handler não detecta o conflito, como ele mantém ambos os nós, é possível que sejam gerados falsos negativos adicionais, uma vez que os nós mantidos podem ser incompatíveis a nível de compilação.

É possível, ainda, que haja conflitos devido a interferências não planejadas, quando foram adicionadas novas chamadas aos métodos. Esse tipo de problema é dificil de ser detectado atualmente, devido ao s3m se basear no uso do *git merge driver* [8], que simplesmente recebe como entrada a tripla de arquivos envolvida no conflito. Seria possível, no entanto, identificar chamadas aos métodos no próprio arquivo e verificar se as modificações resultaram em um falso negativo adicional.

A ideia do presente trabalho é buscar novas formas de lidar com o renaming. São propostas duas novas alternativas, ambas não sendo limitadas

pelo resultado da não estruturada. A primeira busca manter apenas um elemento, ou seja, fazer a união de ambas contribuições sempre que possível. Como o semiestruturado utiliza o não estruturado para fazer a junção das contribuições dos nós relacionados, quando há mudanças mútuas na mesma área, conflitos ainda devem ser reportados para evitar erros adicionais, sendo necessária uma intervenção manual. A outra alternativa, mais simples, busca manter os dois nós das contribuições, de forma a, idealmente, ainda manter um código compilável.

3. Solução

A solução proposta pelo trabalho, então, é o desenvolvimento de formas alternativas de lidar com o renaming. A ideia é que, com isso, o desenvolvedor seja capaz de decidir a melhor configuração de acordo com seu perfil ou situação. São propostas duas novas alternativas, ambas não sendo limitadas pelo resultado da não estruturada.

Chamaremos de MergeMethods a solução que busca unir os nós correspondentes sempre que possível e de KeepBothMethods a solução que mantém ambas as versões.

3.1. MergeMethods

Tem o propósito de resolver os conflitos de renaming sempre que possível, utilizando a similaridade para encontrar os nós correspondentes, independentemente do resultado do não estruturado. A ideia é tentar manter um só elemento, unindo o novo corpo com a nova assinatura. Aos nós correspondentes, é aplicado o merge textual, linha a linha, de forma que se houver alteração na mesma área, conflitos ainda serão reportados.

Além disso, contém parte dos problemas do s3m original, devido a compartilhar parte da lógica. É possível que não haja match por similaridade, devido à medida adotada não ser a ideal, por exemplo. Assim, pode conter falsos negativos adicionais, uma vez que mantendo ambos os nós, pode gerar erro de compilação. Também é possível que ocorra um match errado, devido ao nó mais similar não ser o que corresponde ao método comumente alterado, nesse caso sendo gerado pelo menos um falso positivo adicional, uma vez que o conflito reportado não corresponde ao caso verdadeiro.

A estratégia, então, tem o intuito de detectar e resolver automaticamente um número de conflitos maior do que o s3m original, por não depender do não estruturado. Para um caso como o da Figura 1, por exemplo, onde um

método foi alterado por ambas as partes e movido para áreas distintas, o novo algoritmo é capaz de detectar o conflito, assumindo uma similaridade razoável entre as versões. O caso da Figura 2, onde há uma alteração mútua no corpo, também tem seu conflito devidamente reportado.

3.2. KeepBothMethods

A estratégia KeepBothMethods é a mais simples das duas, com a ideia de manter ambos os nós envolvidos no conflito, com exceção dos casos em que as assinaturas resultantes são iguais. Poderia, além disso, ser verificado se as assinaturas são incompatíveis, questão também presente nos outros algoritmos quando os nós são mantidos. Além do potencial aumento de falsos negativos, é óbvia também a duplicação de código que ele acarreta, o que pode ser proibitivo em certos casos. No entanto, elimina falsos positivos adicionais, uma vez que não reporta conflitos. Esse tipo de estratégia pode ser útil em cenários com poucos renamings, uma vez que evita matches incorretos.

Para ambos os casos citados anteriormente, nas Figuras 1 e 2, ambas as versões dos métodos seriam mantidas e nenhum conflito seria reportado. A princípio, isso não é um grave problema, uma vez que não há erro de compilação. No entanto, a depender do uso e da chamada dos métodos, novos erros podem ter sido introduzidos.

4. Avaliação Empírica

4.1. Dados

Para o estudo, fez-se uso de uma amostra utilizada na avaliação do s3m [1], contendo cenários de merge de projetos na linguagem java. Os critérios de seleção utilizados foram com base nos projetos mais populares e que possuem um grau de diversidade significativo, com critérios como o número de contribuidores, o tamanho da base de código e o domínio [1]. Assim, acredita-se que a amostra utilizada seja suficiente para o estudo em questão.

4.2. Execução e Análise

Para a realização do experimento, a ferramenta foi executada para todos os cenários da amostra, tendo como um dos resultados estatísticas como número de conflitos e número de arquivos com conflito. As estatísticas também foram agrupadas por projeto, de forma a possibilitar a análise de um possível comportamento diferente entre projetos distintos. Além disso, para cada cenário onde pelo menos um das estratégias reportava algum conflito,

estatísticas mais completas são preenchidas, como quantidade de conflitos de renaming, quantidade de conflitos de renaming individual e quantidade de conflitos de renaming mútuo, além de serem salvos os resultados do merge de cada estratégia em arquivos para posterior análise manual[1].

Dentre os casos acima, foram selecionados aleatoriamente 50 casos para análise manual, de forma a melhor entender o comportamento efetivo das estratégias, tanto para certificar que o comportamento desejado estava ocorrendo quanto para detectar situações não imaginadas previamente.

Para a análise, foram estudados os casos onde haviam métodos envolvidos em conflitos por pelo menos uma das estratégias, a partir dos arquivos armazenados. O resultado de cada estratégia foi classificado como verdadeiro positivo (TP), falso positivo (FP), verdadeiro negativo (TN) ou falso negativo (FN), de acordo com os seguintes critérios:

- TP: se acusou conflito entre métodos equivalentes, e houve alteração comum em áreas próximas, sem possibilidade de resolução automática por *3-way-diff*;
- FP: se acusou conflito entre métodos não equivalentes, ou não houve alteração comum entre áreas próximas, podendo haver resolução automática;
- TN: se não acusou conflito, e o resultado gerado não aparenta ter erro de compilação[2];
- FN: se não acusou conflito, e o resultado gerado tem erro de compilação.

5. Resultados e Discussão

Dos 1731 cenários da amostra, foram selecionados aqueles onde houve pelo menos um conflito de renaming detectado por qualquer uma das estratégias do s3m, totalizando 210 cenários e 353 triplas de arquivos. Dessa forma, também foi calculado o número de conflitos de renaming detectados para cada estratégia.

[1]Análise disponível em https://gaabs.github.io/s3m-logs-web/

[2]Note que não há uma garantia de não haver erro de compilação. Essa avaliação foi decidida a partir de uma análise manual do arquivos envolvidos no conflito, sem olhar todos os arquivos do projeto, uma vez que o *merge driver* só recebe a tripla de arquivos como entrada.

Tabela 1: Comparação do número de conflitos de renaming das estratégias

	MergeMethods	s3m	KeepBothMethods
Número de arquivos com conflito de Renaming	332 (100%)	83 (25%)	0 (0%)
Número de conflitos de Renaming	508 (100%)	135 (26.57%)	0 (0%)

Observa-se na Tabela 1, primeiramente, que o MergeMethods reporta cerca de 4 vezes mais conflitos do que a estratégia original. Isso faz sentido, uma vez que passou a detectar conflitos em mais situações, por não depender mais do não estruturado. No entanto, a proporção observada na análise manual não foi a mesma.

É provável, também, que apesar do aumento de conflitos, a maior parte deles sejam conflitos reais (TP), e não falsos positivos. É o que indica a análise manual, onde houve um aumento da proporção de TP em relação à quantidade de FP, para essa nova estratégia.

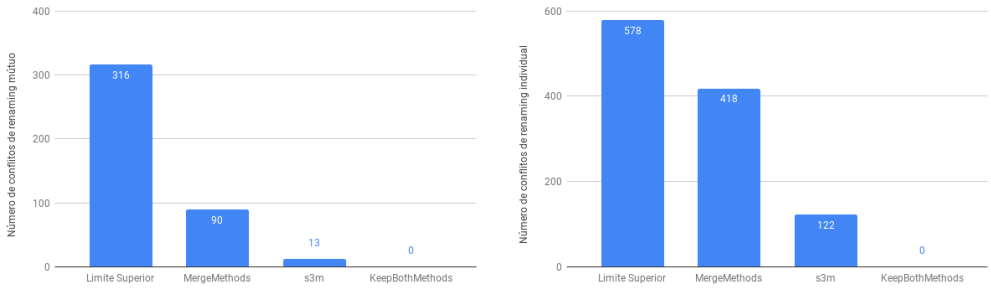

Fig. 3: Comparação do número de conflitos de renaming das estratégias, separando-se em renaming mútuo e renaming individual, respectivamente. Os limites superiores consideram simplesmente a quantidade de nós deletados.

Separando-se, ainda, os resultados entre renaming mútuo e individual, observa-se, na Figura 3, que a maior parte dos conflitos reportados são de renaming individual. Condiz com o esperado, uma vez que é mais incomum que dois desenvolvedores renomeiem o mesmo método [9]. Além disso, a diferença na detecção de conflitos é mais acentuada nos conflitos de renaming mútuo: o MergeMethods tem valor cerca de 7 vezes superior em relação ao s3m original. Isso acontece, principalmente, por conta da exigência de corpos iguais por parte do s3m, como citado anteriormente.

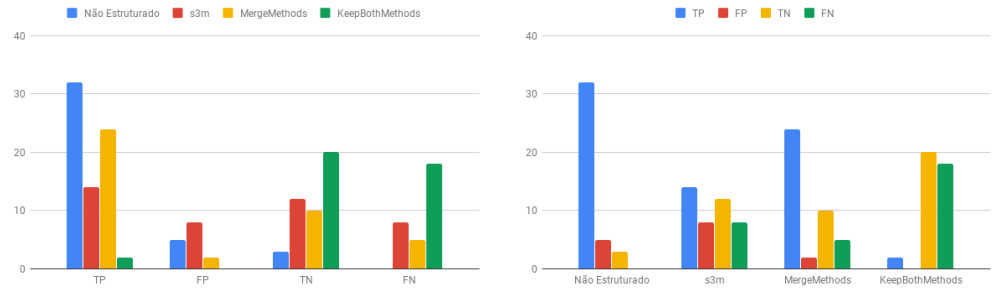

Fig. 4: Análise manual de conflitos

Na análise manual, presente na Figura 4, observa-se que a estratégia original obteve o resultado mais balanceado, tendo, no entanto, um número relativamente alto de falsos positivos e falsos negativos. O MergeMethods, por sua vez, teve uma alta soma de resultados corretos (TP+TN), muito próximo ao resultado do não estruturado, e tendo um número relativamente baixo de resultados incorretos (FP+FN), apesar de não ser melhor do que o não estruturado nesse aspecto. Já o KeepBothMethods não teve nenhum falso positivo, e um número alto de falsos negativos, como esperado.

No entanto, é importante destacar que a análise manual, além ter sido baseada em uma amostra, é propensa a erros. É interessante uma análise de um maior número de casos, de forma a abranger um universo mais representativo e diversificado. Também seria útil uma forma automatizada de comparar os resultados das estratégias, minimizando a necessidade de uma análise manual. Uma alternativa para isto seria utilizar projetos com builds automatizadas, mas além de ser mais difícil de encontrar esse tipo de repositório disponível, quando cruzados com cenários de merge contendo renaming, o espaço amostral é bastante reduzido.

6. Trabalhos Relacionados

Um número grande de estudos tem o objetivo de estudar ou desenvolver formas de entender e melhorar o desenvolvimento colaborativo. O propósito é facilitar o processo de integração, seja melhorando a corretude ou diminuindo o tempo e esforço necessário. O s3m [1], usado como base no presente estudo, foi um avanço em relação às formas de merge, mesmo dentre as alternativas semi estruturadas. No entanto, não teve um foco direcionado a novas formas de lidar com renaming, sendo o presente trabalho um avanço nesse

sentido. Enquanto o s3m original se limita ao não estruturado, as soluções desenvolvidas aqui dão um passo à frente, dando mais opções ao tratamento desse tipo de conflito.

RefactoringMiner [10] é uma solução recente para detecção de refatorações, problema presente em diversas aplicações. Tem o objetivo de detectar refatorações sem thresholds de similaridade. Poderia ser utilizado em conjunto com a ferramenta, de forma a auxiliar na detecção de renamings no s3m ou em outras soluções de merge.

Accioly [9] fez um estudo empirico em cima do s3m, utilizando 70.047 cenários de merge de 123 projetos. O artigo busca entender as carateristicas conflitos de merge observados. O presente trabalho, por outro lado, apesar de abranger um espaço de análise bem menor, tem um foco nos cenários de merge contendo conflitos de renaming.

7. Conclusões

Como alternativa ao merge não estruturado, comparações semi e estruturadas fazem uso da estrutura sintática do código para detectar e solucionar conflitos. Apesar disso, ainda existe muito espaço para melhorias, em particular em casos mais específicos de conflitos na solução semiestruturada.

Dentre os vários possíveis casos de conflitos, foi escolhido o renaming como foco, uma vez que é um dos tipos de refatoração mais frequentes, inclusive com presença considerável em conflitos de merge.

O s3m, uma solução semi-estruturada, possui um tratamento simplificado para lidar com renaming, buscando basicamente não ser pior do que o não estruturado. É possível, por exemplo, que seja reportado um conflito que não deveria ter sido reportado, com match entre elementos que não correspondem realmente; ou que um conflito que não foi reportado quando deveria.

Esse trabalho, então, desenvolveu novos módulos de renaming para serem acoplados ao s3m, de forma a adicionar novas possibilidades de configuração para o usuário, dando a oportunidade de escolha de estratégia a depender da situação. Foram desenvolvidos dois algoritmos, um que sempre mantém os dois nós do conflito, chamado de KeepBothMethods, e outro que busca uní-los, independentemente do resultado do não estruturado, chamado de MergeMethods.

Foram utilizados 1731 cenários de merge na linguagem java, de onde 210 possuíam conflitos de renaming, e 50 foram selecionados para uma análise

manual e qualitativa. Observou-se, com isso, que a estratégia KeepBothMethods reduziu o número de conflitos reportados e teve um aumento significativo de falsos negativos, ou seja, conflitos não acusados e que acarretam falha de integração. A estratégia MergeMethods, por sua vez, teve um aumento na detecção de conflitos verdadeiros em relação ao s3m original, por não depender do não estruturado.

Conclui-se que as soluções propostas são alternativas úteis à resolução de conflitos, em particular ao caso de renaming, dando opções não limitadas ao não estruturado. No entanto, ainda é necessário um maior refinamento para seu uso efetivo na prática. Melhorias ainda podem ser feitas para reduzir o número de falsos positivos, como aperfeiçoamentos nas medidas de similaridade ou possibilitar ao usuário decidir qual nó será escolhido como correspondente no renaming. Para a redução do número de falsos negativos, técnicas como análise sintática do código resultante podem ser úteis. Assim, busca-se trazer uma melhor garantia de integração de código correta. Além disso, é possível ainda que haja uma inteligência por trás da ferramenta de merge que escolha automaticamente a melhor estratégia de acordo com a situação.

Referências

[1] G. Cavalcanti, P. Borba, P. Accioly, Evaluating and improving semistructured merge, Proceedings of the ACM on Programming Languages 1 (2017) 59.

[2] T. Zimmermann, Mining workspace updates in cvs, in: Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on, IEEE, pp. 11–11.

[3] S. Apel, O. Leßenich, C. Lengauer, Structured merge with auto-tuning: balancing precision and performance, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, pp. 120–129.

[4] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering 38 (2012) 5–18.

[5] O. Leßenich, S. Apel, C. Kästner, G. Seibt, J. Siegmund, Renaming and shifted code in structured merging: Looking ahead for precision and

performance, in: Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on, IEEE, pp. 543–553.

[6] T. Mens, A state-of-the-art survey on software merging, IEEE transactions on software engineering 28 (2002) 449–462.

[7] S. Apel, J. Liebig, B. Brandl, C. Lengauer, C. Kästner, Semistructured merge: rethinking merge in revision control systems, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, pp. 190–200.

[8] Praqma, Praqma/git-merge-driver, https://github.com/Praqma/git-merge-driver, 2018.

[9] P. Accioly, P. Borba, G. Cavalcanti, Understanding semi-structured merge conflict characteristics in open-source java projects, Empirical Software Engineering 23 (2018) 2051–2085.

[10] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, D. Dig, Accurate and efficient refactoring detection in commit history, in: IEEE ICSE, volume 2018.