A caminho de uma abordagem baseada em buscas para minimização de conflitos de *merge*

Gleiph Ghiotto[1], Leonardo Murta[1], Marcio Barros[2]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)
Rua Passos da Pátria 158, São Domingos – Niterói – RJ – Brasil

[2]Programa de Pós-graduação em Sistemas de Informação – UNIRIO
Av. Pasteur 458, Urca – Rio de Janeiro – RJ – Brasil

{gmenezes,leomurta}@ic.uff.br, marcio.barros@uniriotec.br

Abstract. *Merge algorithms are widely used to conciliate changes performed in parallel over software artifacts. However, traditional 3-way merge algorithms, based on diff3, are unable to solve some conflicts because they only consider two basic operations: add and remove. These operations cannot accurately describe developer's intention. Moreover, these algorithms ignore intermediate versions, using only the two head versions and a common base version to perform the merge. This paper introduces a search-based approach that aims at minimizing the number of conflicts generated during merge. Our approach considers the developer's intention and the development history during merge, finding alternative merge sequences to reduce conflicts.*

Resumo. *Algoritmos de merge são amplamente utilizados para conciliar alterações realizadas em paralelo sobre artefatos de software. Entretanto, algoritmos tradicionais de merge em três vias, baseados no diff3, não são capazes de solucionar alguns conflitos por considerar apenas duas operações básicas: adição e remoção. Essas operações não expressam precisamente a intenção do desenvolvedor. Além disso, esses algoritmos ignoram versões intermediárias, usando somente as duas versões cabeça e a versão base na execução do merge. Este artigo introduz uma abordagem baseada em buscas que visa minimizar o número de conflitos gerados pelo merge físico. Para tal, a intenção do desenvolvedor e o histórico de desenvolvimento são considerados na execução de merge, encontrando sequências alternativas de merge que reduzam os conflitos.*

1. Introdução

Algoritmos de *merge* são amplamente utilizados para conciliar alterações realizadas em paralelo sobre artefatos de software. Segundo Mens (2002), o suporte para *merge* de software é necessário durante todo o ciclo de desenvolvimento. Essa necessidade surge em especial no desenvolvimento de software complexo em larga escala, onde pode haver um grande número de desenvolvedores distribuídos geograficamente. Este cenário motiva a criação de diversas linhas de desenvolvimento que são frequentemente mescladas, visando consolidar alterações realizadas em paralelo.

A importância dos algoritmos de *merge* se tornou ainda maior com a popularização de sistemas de controle de versão (SCV) distribuídos, como, por exemplo, o Git [Chacon 2009] e o Mercurial [O'Sullivan 2009]. Nesses sistemas, não há

opção da utilização de políticas de controle de concorrência pessimistas, baseadas em bloqueio. Sendo assim, cada desenvolvedor clona para o seu espaço de trabalho uma cópia completa do repositório. Durante seu trabalho, são realizados *commits* locais, que compõem um ramo em relação aos demais desenvolvedores. Ao final, as contribuições locais são reintegradas ao repositório de origem ou a outros repositórios por meio de comandos de *push* e *pull*, requerendo *merges* frequentes.

Quando o algoritmo de *merge* não consegue conciliar automaticamente as contribuições feitas em paralelo, são reportados conflitos para serem tratados manualmente pelos desenvolvedores. Desta forma, o número de conflitos reportados e a complexidade desses conflitos se tornam parâmetros importantes para classificar a qualidade de um algoritmo de *merge*. Por exemplo, os algoritmos de *merge* de duas vias (do inglês, *two-way merge*) consideram somente as duas versões mais recentes de cada linha de desenvolvimento na execução do *merge*. Como consequência, um elevado número de conflitos é gerado por não ser possível diferenciar a intenção de adição por parte de um desenvolvedor da intenção de remoção por parte do outro. Por outro lado, algoritmos de *merge* de três vias (do inglês, *three-way merge*) também levam em conta uma versão base comum aos ramos que devem ser combinados, permitindo uma melhor percepção da intenção dos desenvolvedores, levando a um menor número de conflitos.

Os SCV atuais fazem amplo uso de algoritmos de *merge* de três vias, pois, por guardarem o histórico da evolução do software, sempre têm a versão base disponível. Contudo, esses algoritmos não utilizam todo o histórico de desenvolvimento do projeto e, consequentemente, o conhecimento presente nas versões intermediárias, entre a base e as cabeças dos ramos, é perdido. Essa decisão leva a menos alternativas no momento de resolver automaticamente um conflito. Além disso, os algoritmos de *merge* tradicionais mapeiam as alterações em operações de adição e remoção. Essa estratégia restringe a capacidade de identificação precisa da real intenção dos desenvolvedores.

Diante dessas limitações, é proposta uma abordagem baseada em buscas que tem como objetivo minimizar o número de conflitos gerados pela execução do *merge* físico. Esta abordagem, denominada Kraken, se diferencia das demais por utilizar, na execução do processo de *merge*, o histórico de versões intermediárias de cada ramo, heurísticas para a identificação da real intenção do desenvolvedor e técnicas de busca para identificação de diferentes sequências de aplicação de deltas visando a minimização do número de linhas de código em conflito. Por considerar todo o histórico de versões intermediárias entre a versão base e as cabeças dos ramos, essa abordagem tende a explorar um espaço de busca de ordem fatorial ao número de deltas, representando um cenário propício para aplicação de Engenharia de Software Baseada em Buscas [Harman 2007].

Este trabalho está organizado em seis seções incluindo esta seção de introdução. Na Seção 2 é apresentado um exemplo motivacional que consiste em um cenário de *merge* onde as abordagens tradicionais não são capazes de obter automaticamente uma solução livre de conflitos. Na Seção 3 é dada uma visão geral do estado da arte em *merge* de software e técnicas de busca que podem ser aplicadas para auxiliar na minimização dos conflitos de *merge*. Na Seção 4 é apresentada uma visão geral da abordagem Kraken, caracterizando-a como um problema de otimização, e uma discussão das atividades realizadas por ela para minimizar o número de conflitos. Na

Seção 5 é apresentado como a Kraken soluciona o exemplo motivacional da Seção 2. Finalmente, na Seção 6 são apresentadas as conclusões e discutidos trabalhos futuros.

2. Exemplo motivacional

Conforme discutido na Seção 1, os algoritmos de *merge* tradicionais não são capazes de solucionar, sem conflitos, alguns cenários de *merge* que poderiam ser resolvidos automaticamente. Nesta seção é apresentado um exemplo que reforça a necessidade de uma nova abordagem de *merge*.

Considere a classe `Transformacao`, apresentada na Figura 1, que possui três métodos: um método `celsiusToKelvin`, que está implementado na classe, e outros dois métodos que estão sem conteúdo. Tais métodos foram projetados para realizar transformação de temperatura entre as escalas Kelvin, Fahrenheit e Celsius.

```
class Transformacao {
  public double kelvinToFahrenheit(double kelvin) {
  }
  public double celsiusToKelvin(double celsius) {
    return celsius + 273;
  }
  public double fahrenheitToCelsius(double fahrenheit) {
  }
}
```

Figura 1. Versão base das alterações.

Considere também que, a partir desse artefato mostrado na Figura 1, dois desenvolvedores realizaram alterações em seus ramos. O primeiro desenvolvedor, no ramo 1, realizou duas modificações, como apresentado na Figura 2: Δ_1, que consiste em mover o método `fahrenheitToCelsius` para antes do método `celsiusToKelvin`, e Δ_2, que consiste em adicionar conteúdo ao método `kelvinToFahrenheit`. Em paralelo, o desenvolvedor 2, no ramo 2, realiza duas modificações, como apresentado na Figura 3: Δ_3, que adiciona conteúdo no método `fahrenheitToCelsius`, e Δ_4, que move o método `kelvinToFahrenheit` para depois do método `celsiusToKelvin`.

```
class Transformacao {
  public double kelvinToFahrenheit(double kelvin) {
    return (9 * (kelvin - 273) / 5) + 32;                (Δ2)
  }
  public double fahrenheitToCelsius(double fahrenheit){  (Δ1)
  }                                                      (Δ1)
  public double celsiusToKelvin(double celsius) {
    return celsius + 273;
  }
  public double fahrenheitToCelsius(double fahrenheit) { (Δ1)
  +                                                      (Δ1)
}
```

Figura 2. Versão com alterações do Desenvolvedor 1 (ramo 1).

Analisando a Figura 2 e a Figura 3 é possível visualizar que as alterações realizadas sobre o artefato original resultam em conflitos. Por exemplo, a remoção da linha com o conteúdo " }" pelo Δ_4 e a adição da linha com o conteúdo " return (9 * (kelvin - 273) / 5) + 32;" pelo Δ_2 representa um conflito porque ambos os desenvolvedores editaram a mesma região de um artefato. Este cenário de conflitos pode ser visualizado na Figura 4, onde o Git foi utilizado para realizar *merge* entre os

ramos 1 e 2. Neste cenário é demarcada a versão do *ramo1* e do *ramo2*, pois o algoritmo de *merge* do Git não foi capaz de resolver os conflitos existentes. Deste modo, fica a cargo do desenvolvedor combinar as versões.

```
class Transformacao {
  ~~public double kelvinToFahrenheit(double kelvin) {~~      (Δ4)
  ~~}~~                                                      (Δ4)
  public double celsiusToKelvin(double celsius) {
    return celsius + 273;
  }
  public double kelvinToFahrenheit(double kelvin) {          (Δ4)
  }                                                          (Δ4)
  public double fahrenheitToCelsius(double fahrenheit) {
    return (5 * (fahrenheit -32)) / 9;                       (Δ3)
  }
}
```

Figura 3. Versão com alterações do Desenvolvedor 2 (ramo 2).

```
class Transformacao {
<<<<<<< ramo1
  public double kelvinToFahrenheit(double kelvin) {
    return (9 * (kelvin - 273) / 5) + 32;
  }
  public double fahrenheitToCelsius(double fahrenheit) {
  }
  public double celsiusToKelvin(double celsius) {
    return celsius + 273;
  }
=======
  public double celsiusToKelvin(double celsius) {
    return celsius + 273;
  }
  public double kelvinToFahrenheit(double kelvin) {
  }
  public double fahrenheitToCelsius(double fahrenheit) {
    return (5 * (fahrenheit -32)) / 9;
  }
>>>>>>> ramo2
}
```

Figura 4. Conflito resultante do *merge* dos ramos 1 e 2.

3. Trabalhos relacionados

A resolução de conflitos é tratada por diferentes áreas da computação, como, por exemplo, banco de dados e engenharia de software. Na área de banco de dados, a resolução de conflitos é necessária para combinar transações realizadas em paralelo sobre réplicas de um banco de dados central [Berlage e Genau 1993, Edwards et al. 1997, Kermarrec et al. 2001]. Para realizar essas combinações são exploradas satisfatoriamente diversas estratégias, como, por exemplo, intercalação da ordem das transações, de forma que a combinação das transações seja realizada.

Na área de engenharia de software, foco deste trabalho, a resolução de conflitos é uma preocupação constante em *merge* de software, onde os desenvolvedores realizam alterações sobre artefatos de software que devem ser combinados no futuro. Pesquisas em *merge* de software são realizadas sobre diversos tipos de artefatos, como, por exemplo, modelos [Koegel et al. 2010, Murta et al. 2008], arquivos binários [Silva Junior et al. 2012] e arquivos texto [Mens 2002, Shen e Sun 2004]. O escopo deste artigo é voltado para resolução de conflitos em *merges* textuais, independentemente da estrutura sintática utilizada. Esse tipo de *merge* é de especial relevância por ser

genérico, podendo estar incorporado em SCV e apoiar projetos de software independentemente da linguagem adotada. Portanto, essa seção trata desse tipo de *merge*.

Como discutido anteriormente, os SCV atuais empregam de forma extensiva o *merge* textual de duas e três vias. O *merge* de duas vias [Mens 2002] é capaz de reconhecer que dois artefatos são diferentes, extrair o delta que representa essa diferença e apoiar na combinação dos dois artefatos. Contudo, nenhum conhecimento a mais pode ser extraído devido à ausência de um referencial sobre a forma que ocorreu a evolução. Para solucionar esse problema, o *merge* de três vias [Mens 2002] utiliza um ancestral comum às versões que serão combinadas. Neste caso, o ancestral comum serve como parâmetro para que as operações de adição ou remoção sejam mapeadas. Por exemplo, supondo que uma linha de código exista no ancestral e em uma das versões, mas não exista mais na outra, então esta linha é marcada como removida. Por outro lado, a adição pode ser identificada por uma linha de código que não exista no ancestral e também não exista em uma das versões, mas passou a existir na outra versão.

Com o intuito de melhorar as abordagens de *merge* tradicionais Shen e Sun (2004) desenvolveram uma abordagem baseada em operações que realiza o *merge* de dois ramos. Para tal, é utilizada uma correção no local que haveria conflito, permitindo a combinação de ambas as alterações em diferentes ordens. Apesar dessa abordagem ser capaz de conciliar alguns conflitos que são reportados por abordagens de *merge* de três vias, ela pode gerar falsos negativos em algumas situações. Esses falsos negativos, que ocorrem quando conflitos reais não são reportados, são causados quando as edições em paralelo têm o mesmo propósito. Por exemplo, caso haja na versão base uma expressão “a = b;”, que foi editada em paralelo para “a = 1 + b;” e para “a = b + 1”, nenhum conflito seria reportado e o resultado final seria “a = 1 + b + 1;”, o que provavelmente é equivocado. Além disso, por não capturar a real intenção dos desenvolvedores durante a evolução, esta abordagem não seria capaz de resolver conflitos como o mostrado na Seção 2.

Diante do exposto, é possível identificar que o histórico de desenvolvimento dos projetos e a intenção do desenvolvedor são recorrentemente ignorados na literatura. Ao descartar o histórico de desenvolvimento, perde-se o conhecimento sobre as pequenas mudanças que, quando combinadas, geram a alteração como um todo. Essas pequenas mudanças refletem as intenções originais do desenvolvedor. Por outro lado, ao mapear a intenção do desenvolvedor somente em operações de adição e remoção, as abordagens perdem conhecimento que poderia ser usado no processamento do *merge*. Além disso, é válido ressaltar que as abordagens descritas nesta seção não utilizam qualquer técnica de busca.

4. Kraken

A Kraken é uma abordagem baseada em busca que tem como objetivo minimizar o número de conflitos reportados para o desenvolvedor na execução do *merge*. Para isso, são utilizadas técnicas de buscas/otimização, heurísticas para a resolução de conflitos e o histórico de desenvolvimento para que as decisões quanto a existência ou não de conflitos possam ser tomadas com base na intenção do desenvolvedor. O problema consiste na geração de diversas sequências de deltas que serão avaliadas quanto a sua ordem de aplicação objetivando alcançar o menor número de conflitos.

Na Tabela 1 é apresentado um levantamento realizado em 6 projetos de código aberto que justificam a aplicação de uma técnica de busca dado o tamanho dos espaços de busca, ou seja, o número de sequências que podem ser geradas. Esta tabela possui dados como: tamanho médio de cada ramo, dado em função do número de deltas, o desvio padrão no tamanho dos ramos e os cenários de *merge* que possuem maior número de deltas para serem combinados. Além disso, é calculado o tamanho do espaço de busca para cada um desses cenários complexos.

Tabela 1. Análise de projetos *open-source*.

Projeto	Ramos		Cenário mais complexo		
	Tamanho médio	Desvio padrão	Ramo 1	Ramo 2	Espaço de busca
Git	171,3	750,6	4734	4612	$9{,}93 \times 10^{2810}$
Jenkins	22,4	75,5	76	2623	$1{,}08 \times 10^{149}$
Linux	76,8	185,8	1704	1621	$4{,}13 \times 10^{998}$
Perl	16,6	87,6	214	60	$2{,}05 \times 10^{61}$
Storm	10,9	19,4	95	19	$7{,}99 \times 10^{20}$
Voldemort	10,4	20,4	205	69	$7{,}90 \times 10^{65}$

A Kraken recebe como entrada um conjunto de ramos de um SCV que são processados até que os deltas possam ser aplicados a uma versão base. Esse processamento inclui a identificação dos deltas, o enriquecimento das operações e a aplicação de mecanismos de busca para a identificação das sequências de deltas com o menor número possível de conflitos. No restante dessa seção é apresentada a formalização do problema e são detalhadas as atividades que compõem a Kraken.

4.1. Formalização do problema

Conforme discutido anteriormente, a Kraken recebe como entrada um conjunto de ramos que, por sua vez, é formado por uma sequência de deltas, como apresentado na Figura 5. Desta forma, considere que n é o número de ramos, $R = \{r_i \mid 1 \leq i \leq n\}$ o conjunto de todos os ramos, e $r_i = (\Delta_{i,1}, \Delta_{i,2}, ..., \Delta_{i,t(ri)})$ a tupla que representa a sequência de deltas de um ramo r_i de comprimento $t(r_i)$. O problema em questão consiste na geração de uma sequência de deltas m, que contém todos os deltas de todos os ramos, respeitando a ordenação imposta por cada ramo, tal que o número de linhas de código em conflito, calculado pela função f(m), seja minimizado.

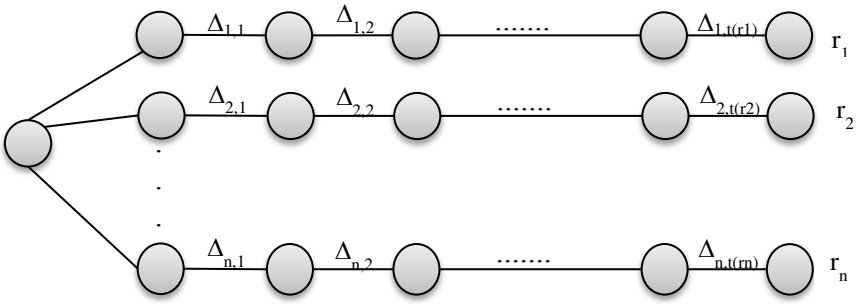

Figura 5. Ilustração dos ramos e deltas.

Para gerar as sequências de deltas candidatas para solucionar o problema, os ramos são combinados de modo que os deltas sejam preservados na mesma ordem em que aparecem em cada ramo. Ou seja, uma sequência de deltas (..., $\Delta_{i,p}$, ..., $\Delta_{i,q}$, ...) é válida se, e somente se, $\forall r_i \in R, 1 \leq p < q \leq t(r_i)$. Por exemplo, suponha um cenário composto por dois ramos, r_1 e r_2, onde r_1 é composto por $\Delta_{1,1}$ e $\Delta_{1,2}$, e r_2 é composto por

$\Delta_{2,1}$ e $\Delta_{2,2}$ é possível a geração de 6 sequências de deltas válidas: ($\Delta_{1,1}$, $\Delta_{1,2}$, $\Delta_{2,1}$, $\Delta_{2,2}$), ($\Delta_{1,1}$, $\Delta_{2,1}$, $\Delta_{1,2}$, $\Delta_{2,2}$), ($\Delta_{1,1}$, $\Delta_{2,1}$, $\Delta_{2,2}$, $\Delta_{1,2}$), ($\Delta_{2,1}$, $\Delta_{1,1}$, $\Delta_{2,2}$, $\Delta_{1,2}$), ($\Delta_{2,1}$, $\Delta_{1,1}$, $\Delta_{1,2}$, $\Delta_{2,2}$) e ($\Delta_{2,1}$, $\Delta_{2,2}$, $\Delta_{1,1}$, $\Delta_{1,2}$).

Esse problema possui espaço de busca com uma tendência de crescimento fatorial. O problema está associado a uma sucessão de combinações que são aplicadas de modo que os deltas possam ser alocados em locais diferentes da sequência e o número de combinações possíveis é dado pela fórmula $\frac{[t(r_1)+t(r_2)+\cdots+t(r_n)]!}{t(r_1)!t(r_2)!\ldots t(r_n)!}$. Alguns exemplos estão representados na Tabela 1.

4.2. Resolução do problema

Para atingir o objetivo de minimizar o número de conflitos gerados, a Kraken foi organizada em cinco atividades, conforme representado no diagrama de atividades da Figura 6. Cada atividade possui objetivos específicos que serão descritos no decorrer desta seção.

Figura 6. Visão geral da Kraken.

A Kraken recebe como entrada ramos que devem ser mesclados e a primeira atividade consiste em identificar todos os deltas presentes neles. Nesta atividade é utilizado um algoritmo de *diff* tradicional, aplicado para identificar as diferenças entre versões consecutivas (Figura 7), neste exemplo é possível visualizar o *diff* resultante do Δ_4 (Figura 3). Ou seja, é feita uma varredura entre as versões, que são analisadas duas a duas, para identificação dos deltas, em modo texto, que serão interpretados durante a atividade de tradução.

```
@@ -1,9 +1,9 @@
 class Transformacao {
-  public double kelvinToFahrenheit(double kelvin) {
-  }
   public double celsiusToKelvin(double celsius) {
     return celsius + 273;
   }
+  public double kelvinToFahrenheit(double kelvin) {
+  }
   public double fahrenheitToCelsius(double fahrenheit) {
   }
 }
```

Figura 7. Delta em modo texto.

A tradução é a atividade responsável por dar semântica aos deltas encontrados pela diferenciação. Os deltas identificados na atividade anterior não possuem semântica alguma para a Kraken, por se tratarem apenas de texto. Portanto, durante a tradução são identificadas as linhas que foram alteradas e essas linhas são traduzidas para operação de adição ou remoção, ainda conforme os algoritmos de *diff* tradicionais são capazes de identificar.

Com base nas operações de adição e remoção obtidas durante a tradução, a atividade de enriquecimento é executada para que a intenção do desenvolvedor seja mapeada de forma mais precisa. A principal contribuição desta atividade é identificar operações de edição, quando uma linha tem o seu conteúdo alterado, e movimentação,

quando um conjunto de linhas é movido de uma região para outra. Portanto, nessa atividade são realizadas buscas por operações que removem e adicionam o mesmo conteúdo, caracterizando uma movimentação de um bloco de linhas, e operações que substituem o conteúdo (remoção e adição na mesma linha), caracterizando uma edição.

De posse de um conjunto de deltas enriquecidos, a atividade de otimização é realizada. Nessa atividade são geradas sequências de deltas que combinam os ramos, sempre respeitando a ordem original de cada ramo. Em seguida, os resultados do *merge* são avaliados visando a minimização do número de linhas de código conflitantes. Como o número de sequências possíveis pode ser muito grande, são utilizados algoritmos de busca, como, por exemplo, algoritmos genéticos, *hill climbing* e *simulated annealing* [Glover e Kochenberger 2003], para explorar o espaço de busca de forma eficiente, resultando em sequências que se aproximem do resultado ótimo.

O resultado da atividade de otimização consiste em uma sequência com o menor número possível de conflitos dentre as alternativas investigadas. Essa sequência é materializada através da atividade de reconstrução. Logo, dada a sequência de deltas e uma versão base (ancestral comum dos ramos) a atividade de reconstrução aplica os deltas na ordem escolhida sobre a versão base. Deste modo, o desenvolvedor recebe a versão final do *merge* com o menor número de conflitos que a Kraken foi capaz de identificar.

5. Exemplo de utilização

Na Seção 2 é discutido um exemplo onde as abordagens tradicionais de *merge* não conseguem realizar a combinação de forma automática e reportam conflitos. Alguns conflitos existentes em abordagens tradicionais ocorrem por deficiências no mapeamento da intenção do desenvolvedor e falta de heurísticas para a resolução de conflitos. Portanto, nessa seção o exemplo da Seção 2 é retomado para ilustrar como a Kraken é capaz de solucionar conflitos. Nesse exemplo, são ilustrados os resultados após cada uma das etapas e são destacadas as atividades de enriquecimento e otimização.

Na Seção 2 é possível identificar que os deltas estão todos escritos em função de adição e remoção. Porém, existem operações que podem ser reescritas de forma mais precisa, representando melhor a intenção do desenvolvedor. Aplicando o enriquecimento sobre este exemplo, o Δ_1 passa a ser mapeado como uma movimentação das linhas do método `fahrenheitToCelsius` do local original, abaixo do método `celsiusToKelvin`, para as linhas imediatamente acima deste método. Essa detecção é possível pois o conteúdo removido é igual ao conteúdo adicionado. Por sua vez, Δ_2 é mantido como uma adição, dado que não possui operações que se encaixam nos perfis de edição e nem de movimentação. No outro ramo, os dois deltas também são enriquecidos. Enquanto Δ_3 é mantido como uma adição, o Δ_4, passa a ser mapeado como uma movimentação das linhas do método `kelvinToFahrenheit` para baixo do método `celsiusToKelvin`.

Com as operações enriquecidas, a atividade de otimização gera as sequências possíveis e calcula a função de *fitness* para cada uma delas. Durante a otimização, são geradas diversas sequências para buscar por aquela que possua o menor número de conflitos possível que, neste caso, é a sequência (Δ_3, Δ_1, Δ_2, Δ_4). Para que as alterações sejam aplicadas corretamente, as operações são reescritas respeitando o contexto gerado

pelos outros deltas e decorrentes do enriquecimento. Ou seja, cada um dos deltas é aplicado sobre o resultado da aplicação do delta anterior.

No caso da sequência ($\Delta_3, \Delta_1, \Delta_2, \Delta_4$), a operação de movimentação é responsável por mover linhas adicionadas em outros ramos para que o resultado obtido reflita a intenção do desenvolvedor. O processamento dessa sequência inicia aplicando Δ_3, que consiste na adição de uma linha de código sobre a versão base. Como esse é o primeiro delta a ser aplicado, ele não sofre qualquer interferência de outros deltas. Em seguida é aplicado o Δ_1, que consiste em mover o bloco onde o Δ_3 foi inserido. Portanto, esse delta é reescrito para que o conteúdo adicionado por Δ_3, que está entre o início e o fim do bloco definido em Δ_1, também seja também movido. Após a aplicação do Δ_3, o Δ_2 é aplicado, que também consiste na adição de uma linha de código. Embora já tenham acontecido alterações sobre a versão base, essa operação não é afetada, pois está fora raio de ação dos deltas anteriores. Finalmente, Δ_4 consiste na movimentação do método `kelvinToFahrenheit`, e é alterado para incluir a linha de código adicionada pelo Δ_2 e para adaptar o destino da movimentação para compensar a ação dos deltas anteriores. Com o resultado obtido no passo de otimização, o resultado do *merge* é produzido, como mostrado na Figura 8.

```
class Transformacao {
  public double fahrenheitToCelsius(double fahrenheit) {
    return (5 * (fahrenheit -32)) / 9;
  }
  public double celsiusToKelvin(double celsius) {
    return celsius + 273;
  }
  public double kelvinToFahrenheit(double kelvin) {
    return (9 * (kelvin - 273) / 5) + 32;
  }
}
```

Figura 8. Resultado com utilização da Kraken.

6. Conclusão

Neste artigo é apresentada a Kraken, uma abordagem baseada em buscas para encontrar a melhor sequência de aplicação dos deltas de forma que o número de conflitos seja minimizado. Este artigo apresenta como contribuições a formalização do problema do *merge* e uma estrutura, organizada em atividades, que é utilizada para atingir o objetivo de minimização do número que conflitos na execução do *merge*. Nessa estrutura é possível ressaltar as atividades de enriquecimento e otimização, que são inovadoras no que tange a resolução de conflitos de *merge*.

Kraken está sendo implementada utilizando a linguagem Java e usando como fonte de dados o SCV Git. Neste momento estão sendo realizados experimentos iniciais sobre projetos de código aberto, considerando *merges* que de fato ocorreram. Como trabalhos futuros serão realizados (1) experimentos confrontando o resultado obtido pela Kraken com resultados de abordagens tradicionais e com os gabaritos, *merges* reais dos repositórios; (2) experimentos para identificar o algoritmo de busca que resolve melhor este problema, iniciando por algoritmos como *Hill Climbing* e evoluindo para Algoritmos Genéticos ou Colônia de Formigas, possibilitando a escolha do algoritmo que utilize menos recurso computacional, por exemplo; e (3) a extensão da abordagem para considerar a sintaxe de linguagens de programação na solução de conflitos, caracterizando assim como *merge* sintático.

Agradecimentos

Os autores gostariam de agradecer à CAPES, ao CNPq e à FAPERJ pelo auxílio financeiro.

Referências

Berlage, T. and Genau, A. (1993). A framework for shared applications with a replicated architecture. In *Proceedings of the 6th annual ACM symposium on User interface software and technology*. , Symposium on User Interface Software and Technology (UIST).

Chacon, S. (2009). *Pro Git*. 1. ed. Berkeley, CA, USA: Apress.

Edwards, W. K., Mynatt, E. D., Petersen, K., et al. (1997). Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*. , Symposium on User Interface Software and Technology (UIST).

Glover, F. W. and Kochenberger, G. A. [Eds.] (2003). *Handbook of Metaheuristics*. 1. ed. New York, Boston, Dordrecht, London, Moscow: Springer.

Harman, M. (2007). The Current State and Future of Search Based Software Engineering. In *2007 Future of Software Engineering*.

Kermarrec, A.-M., Rowstron, A., Shapiro, M. and Druschel, P. (2001). The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. , Symposium on Principles of Distributed Computing (PODC).

Koegel, M., Herrmannsdoerfer, M., Von Wesendonk, O. and Helming, J. (2010). Operation-based conflict detection. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*. , International Workshop on Model Comparison in Practice (IWMCP).

Mens, T. (2002). A State-of-the-Art Survey on Software Merging. *IEEE transactions on software engineering*, v. 28, n. 5, p. 449–462.

Murta, L. G. P., Corrêa, C. K. F., Prudêncio, J. G. and Werner, C. M. L. (2008). Towards odyssey-VCS 2: improvements over a UML-based version control system. In *International Workshop on Comparison and Versioning of Software Models (CVSM)*.

O'Sullivan, B. (2009). *Mercurial : the definitive guide*. 1. ed. Sebastopol CA: O'Reilly Media.

Shen, H. and Sun, C. (2004). A complete textual merging algorithm for software configuration management systems. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*.

Silva Junior, J. R., Pacheco, T., Clua, E. and Murta, L. (2012). A GPU-based Architecture for Parallel Image-aware Version Control. In *European Conference on Software Maintenance and Reengineering (CSMR)*.