An approach for updating forks against the original project

Arthur Roberto Marcondes, Ricardo Terra
arthur.marcondes@estudante.ufla.br,terra@ufla.br
Universidade Federal de Lavras
Lavras, MG

ABSTRACT

Several software projects start from an existing project. This practice, in the VCS ecosystem, is called fork. For instance, the Bootstrap project, initially developed on Twitter, today has more than 68,000 forks, which indicates that several projects started from the Bootstrap source code at a certain moment and are being customized. The problem occurs when customized projects want to obtain updates from the original project, i.e., new features, bug fixes, etc. The merge of the source code between the original and the customized projects usually generates conflicts that need human resolution. More important, the resolution of those conflicts might not be trivial and poses an arduous task for developers. This article, therefore, proposes an approach for updating forks against the original project where features are modularized, documented, traceable, and can be reused. We claim that the such task can no longer be carried out on an ad hoc basis. In a nutshell, instead of modify the method `foo` from the original project, the developer implements it locally and specifies, using one of the nine instructions of the proposed DSL, something like "replace the `foo` method with local implementation". We have developed a tool that automates our approach and conducted an evaluation on a large-scale real-world project that is regularly updated against your original project.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**; *Software configuration management and version control systems*; Software evolution; Maintaining software.

KEYWORDS

Evolução de software, desenvolvimento colaborativo, *merge*, *fork*, conflitos de mesclagem.

ACM Reference Format:
Arthur Roberto Marcondes, Ricardo Terra. 2020. An approach for updating forks against the original project. In *SBES'20: Simpósio Brasileiro de Engenharia de Software, Outubro 23–27, 2020, Natal, RN*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

SBES'20, Outubro 23–27, 2020, Natal, RN

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUÇÃO

Os Sistemas de Controle de Versão (VCSs) proveem aos desenvolvedores técnicas e ferramentas para lidar com a criação e evolução de sistemas de software [13, 15]. Este artigo é centrado na prática de *fork*, a qual cria novos projetos (a.k.a., *projetos customizados*) a partir da cópia de projetos existentes (a.k.a., *projetos originais*) [7, 10, 19]. Modernos VCS, como `Git` e `Subversion`, aliados ao surgimento de plataformas web de desenvolvimento como o `GitHub` fomentaram o surgimento de novos sistemas de software independentes baseados em *forks* de sistemas populares [2, 20]. Por exemplo, o projeto Bootstrap, inicialmente desenvolvido no Twitter e atualmente disponível no GitHub, possui hoje mais de 68 mil *forks*.

Apesar de possuírem evolução própria, o *problema* abordado neste artigo é quando os sistemas customizados querem se manter atualizados frente aos projetos originais, i.e., querem obter novas *features*, correções de *bugs*, melhorias de desempenho, etc. A mesclagem do código do projeto original (*PO*) no projeto customizado (*PC*) pode gerar inúmeros conflitos [8, 11, 12, 16] em decorrência, por exemplo, de (i) o PC criou uma nova classe de domínio em uma tela, porém o PO também criou só que de forma diferente, (ii) o PC alterou o tamanho de um campo, porém o PO também alterou mas para um valor diferente e (iii) o PC corrigiu um *bug*, porém o PO também corrigiu só que de forma diferente.

Embora seja algo rotineiro no desenvolvimento de software, solucionar conflitos de mesclagem de código pode ser considerada uma tarefa árdua para desenvolvedores [3, 12, 16]. Além de custosa, quando a resolução dos conflitos é mal executada, leva a erros de integração e interrupção do fluxo de trabalho, comprometendo a eficiência e o cronograma do projeto [12]. Estudos apontam que essa realidade está presente em aproximadamente 19% das mesclagens de código [12], podendo chegar a percentuais maiores (34% a 54%) em função das características dos projetos [8]. No cenário deste artigo, acredita-se em um alto percentual de conflitos, uma vez que são dois projetos independentes com diferentes equipes trabalhando de forma paralela e, mais importante, a equipe do PO sequer tem conhecimento do que vem sendo customizado em seus *forks*.

Este artigo, portanto, propõe uma abordagem sistemática e com o mínimo de intervenção humana para conduzir o desenvolvimento de *forks* que devem ser atualizados frente ao projeto original, de uma forma **não *ad hoc***, onde *features* são modularizadas, documentadas, rastreáveis e podem ser reutilizadas. Cada customização é composta de um arquivo descritor – o qual fomenta modularidade, documentação e rastreabilidade – e classes locais que implementam as customizações de forma não invasiva. Conforme ilustrado na Figura 1, o processo de atualização é simples: após trazer uma cópia exata do código fonte atual do projeto original, são aplicadas as customização especificadas para o sistema.

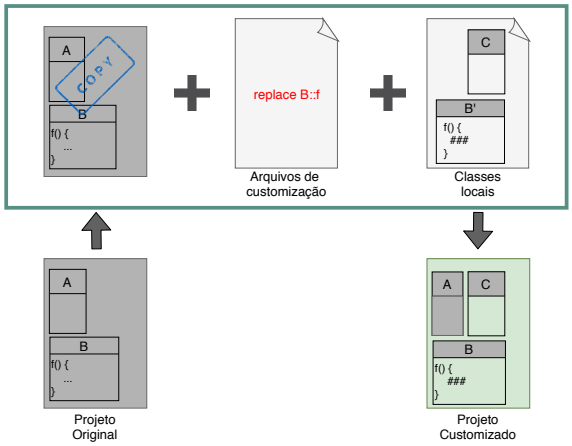

Figura 1: Abordagem para atualização de *forks* frente ao projeto original

A ideia central é que o projeto customizado seja desenvolvido como um "conjunto de alterações que devem ser aplicadas no projeto original", i.e, PC = PO + customizações. Por exemplo, ao invés de alterar o corpo de um método **`foo`** do projeto original, o desenvolvedor o implementa localmente e especifica, por meio de uma das nove instruções da DSL proposta, algo como "substitua o método **`foo`** pela implementação local".

Uma ferramenta que automatiza a abordagem proposta foi desenvolvida para conduzir uma avaliação real em um projeto de software de grande porte frequentemente atualizado frente ao seu projeto original. Como resultado mais relevante, a abordagem conseguiu reduzir 752 conflitos que ocorreram durante 42 atualizações em um período de quatro anos.

O restante deste artigo está estruturado como a seguir. A Seção 2 introduz conceitos fundamentais, como `Git`, *branch* e *fork*. A Seção 3 descreve a abordagem proposta, apresentando as instruções de customização idealizadas. A Seção 4 apresenta a ferramenta implementada. A Seção 5 discute os resultados da avaliação da abordagem proposta em um sistema real de grande porte frequentemente atualizado frente ao seu projeto original. Por fim, a Seção 6 apresenta trabalhos relacionados e a Seção 7 conclui.

2 *BACKGROUND*

Esta seção provê os conceitos fundamentais ao correto entendimento do artigo.

Git: O `Git` é um dos mais populares VCS distribuídos [1], tendo um sistema de ramificação leve e fácil de usar [9]. É amplamente adotado em projetos de pequeno a grande porte por profissionais independentes a grandes equipes de desenvolvimento.

Branch: Uma *branch* é uma ramificação do projeto de software em um VCS. Pode ser a ramificação principal de trabalho (*master*) ou uma ramificação criada para desenvolvimento de uma funcionalidade em paralelo. A Figura 2 apresenta uma visão simplificada de uma *branch* criada para implementação de uma *feature* qualquer.

Observe que a ramificação *master* refere-se à linha de desenvolvimento principal. No tempo t_1, uma nova ramificação *feature* é criada, trabalhada em paralelo e mesclada na ramificação principal

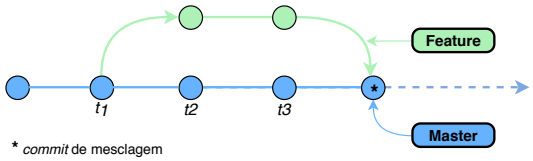

Figura 2: Uma *branch* para desenvolvimento de uma *feature* em paralelo à ramificação principal

somente no tempo t_4. Conforme Spinellis [18], essa é uma prática comum pois permite que seja criada, a qualquer momento, uma cópia privada completa e instantânea do repositório de trabalho atual.

Fork: *Fork* é o conceito de se criar uma cópia completa de um projeto de software [19], normalmente com dois objetivos distintos: (i) possibilitar o trabalho local de um desenvolvedor que vai atuar na correção e evolução daquele sistema ou (ii) criar um novo sistema de software a partir de um sistema já existente. Popularmente e também neste artigo, o termo *fork* é normalmente empregado no segundo contexto [17].

Um projeto de software (a.k.a., *projeto customizado*) criado a partir do *fork* de um *projeto original* tem total independência para evoluir conforme suas necessidades [20]. No entanto, em certos cenários, o projeto customizado necessita aproveitar novas características incorporadas ao projeto original, bem como evoluções e correções de *bugs* [19]. Isso representa uma vantagem ao projeto independente, pois implica em economia de recursos.

Diante disso, deve-se realizar a mesclagem do código do projeto original no projeto customizado, o que pode resultar em conflitos de mesclagem, isto é, como mesclar um mesmo trecho de código alterado tanto no projeto original como no customizado? Se tais conflitos ocorrem rotineiramente em mesclagem de código dentro de um mesmo projeto, conjectura-se que tais conflitos ocorram com mais frequência quando se trata de *forks* que envolvem dois projetos independentes.

3 ABORDAGEM DE CUSTOMIZAÇÃO

A abordagem de customização proposta neste artigo é voltada exclusivamente a projetos customizados que querem se manter atualizados frente ao projeto original. A ideia central é a de que um projeto customizado deve ser desenvolvido como um "conjunto de alterações que devem ser aplicadas no projeto original". Esta seção indica como definir as *features* e como descrever as alterações por meio de instruções de customização. Mais importante, esta seção provê evidências de que a abordagem proposta não só evita conflitos de mesclagem como também promove modularidade, documentação, rastreabilidade e reúso.

3.1 *Exemplo motivador*

A Figura 3 ilustra um sistema web de gerenciamento de academia desenvolvido na linguagem Python sob o *framework* `Django`[1]. Esse sistema – que é utilizado para ilustrar a abordagem no decorrer desta seção – possui cadastros essenciais para atender ao seu propósito, como cadastro de alunos, matrículas, turmas e mensalidades, etc.

[1]Disponível em https://www.djangoproject.com/.

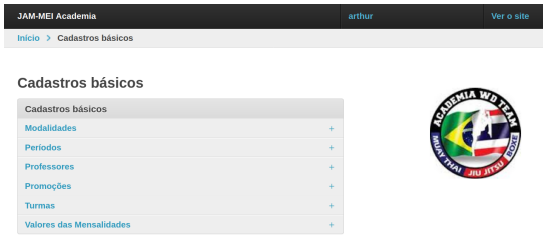

Figura 3: Tela do módulo Cadastros básicos

Para ilustrar a definição de *features*, será criado um projeto customizado a partir do *fork* do projeto motivador. Nesse projeto customizado, idealizou-se três *features*:

#*F*1: *Customização do leiaute*: troca do logotipo da academia e cores de elementos em tela.

#*F*2: *Inclusão de recepcionista*: criação de um novo modelo de dados **`Recepcionista`**, com todas as funções de CRUD[2], dentro do módulo do sistema denominado "Cadastros básicos".

#*F*3: *Recepcionista em matrícula*: alteração do modelo de dados **`Matricula`**, dentro do módulo "Alunos", com a inclusão da informação sobre a recepcionista responsável pela efetivação de uma matrícula.

3.2 Definição de *features*

Uma *feature* pode ser definida como uma funcionalidade de um sistema de software que resolve algum problema do mundo real ou que entrega algum benefício aos seus usuários. Na abordagem proposta, customizações a serem realizadas em um sistema devem ser organizadas em *features*.

Um *arquivo de customização*, portanto, define uma única *feature* e deve possui as seguintes diretrizes:

- **`#feature`**: identificação única da *feature*;
- **`#description`**: informações sobre a customização, promovendo documentação; e
- **`#dependsOn`**: identificação das *features* que são obrigatórias para essa *feature*, promovendo relações de dependência.

Além do supracitado, essa definição promove (i) modularidade, pois um arquivo de customização engloba todas as alterações que a *feature* requer e (ii) reúso, pois esse arquivo pode ser utilizado em outros projetos customizados. Isso promove também tomadas de decisão estratégicas. Por exemplo, se diversas customizações dependem de uma determinada customização `X`, modificá-la pode não ser uma boa ideia ou deve ser feita com cautela por meio de programação aos pares, por exemplo.

Exemplo motivador: A Listagem 1 ilustra a definição da *feature* #*F*3. Como pode-se observar, além do nome e de uma descrição, existe a definição de uma dependência da *feature* #*F*3 com a #*F*2.

```
1 #feature 003-recepcionista-matricula
2 #description Altera a Matricula incluindo
3              recepcionista responsável
4 #dependsOn 002-recepcionista
```

Listagem 1: Fragmento do arquivo de customização F#3

[2]Na Indústria é o nome popularmente dado a uma tela que realiza as operações básicas *Create, Retrieve, Update e Delete* em uma classe de domínio.

As *features* #*F*1 e #*F*2 são bem simples e possuirão apenas as diretivas **`#feature`** e **`#description`**.

3.3 *Workflow* de desenvolvimento

Um potencial *workflow* para aplicação da abordagem proposta em um cenário real de desenvolvimento de software seria:

(a) dada uma solicitação de desenvolvimento, o desenvolvedor busca (ou cria) o arquivo de customização a qual a *feature* referente àquela solicitação pertence;
(b) implementa o código local necessário para atender à solicitação (a ser integrado pela ferramenta) e cria as instruções necessárias no arquivo de customização;
(c) busca o código atual do sistema original em uma nova *branch* e executa a ferramenta a partir dos arquivos de customização com o uso dos arquivos locais do projeto customizado; e
(d) avalia se o código resultante da customização (i) compila e executa sem erros, (ii) se implementa a solicitação de desenvolvimento e (iii) se passa por rotinas de testes e inspeções já estabelecidas no processo de desenvolvimento.

Os arquivos de customização e arquivos locais do projeto customizado devem estar sob o controle de versão, sendo possível modificá-los ao longo do tempo. As customizações são "codificadas" pelos desenvolvedores, ou seja, a abordagem não identifica e deriva as instruções de customização de forma automática.

3.4 Instruções de customização

Um arquivo de customização inclui pelo menos uma instrução de customização. Essas instruções especificam o que deve ser realizado sobre o código fonte do projeto original de tal forma a transformá-lo no projeto customizado.

3.4.1 Levantamento de instruções. O conjunto de instruções foi levantado empiricamente pela análise de diversos conflitos de *merge* durante um ano de desenvolvimento de um *fork* de um sistema real de administração pública.

SUAP (Sistema Unificado de Administração Pública): Iniciado em 2007, é um projeto desenvolvido e mantido pelo IFRN (Instituto Federal do Rio Grande de Norte). Trata-se de um sistema de grande porte com 2.600 arquivos .py distribuídos ao longo de 80 módulos, com 330.000 linhas de código (excluídas linhas em branco e linhas de comentário), além de outros arquivos típicos de uma aplicação web, como arquivos CSS, JavaScript e HTML. Em 2014, passou a ser utilizado e customizado pelo IFSULDEMINAS (Instituto Federal do Sul de Minas). Desde então, diversas customizações foram realizadas no código local desenvolvido pela equipe do IFSULDEMINAS, embora obtendo as atualizações realizadas a partir do código do ramo principal do projeto no IFRN.

Simulação de conflitos de merge: Adotou-se arbitrariamente a data de um ano antes do início dos experimentos (17/09/2018). Nessa data, existiam cerca de 460 arquivos com diferenças entre os projetos customizado e original. Os *branches* de cada projeto foram retrocedidos para essa data e tentou-se uma operação de fusão do código original no código customizado. Como resultado, obteve-se mais de 200 trechos com conflitos.

Análise dos conflitos: A análise individual de cada um dos 200 conflitos expôs a dificuldade da resolução manual. O primeiro autor deste artigo despendeu horas na análise e resolução dos conflitos. Assim, cada diferença identificada foi utilizada para idealização e especificação de um conjunto de nove instruções em DSL, conforme reportado na Tabela 1, que ajudassem a atingir o objetivo de transformar o código do projeto original no código customizado.

Tabela 1: Instruções em linguagem DSL

#	Instrução	Semântica
1	replace file	Substitui determinado artefato de software (arquivo).
2	replace unit	Substitui o corpo de determinada classe, função ou método.
3	replace string	Substitui determinada *string* em determinada classe, função, método ou arquivo.
4	add @annotation	Adiciona uma *annotation* em determinada classe, função ou método.
5	add unit	Cria determinada classe, função ou método em determinado arquivo fonte.
6	remove @annotation	Remove determinada *annotation* de determinada classe, função ou método.
7	remove unit	Remove determinada classe, função ou método de um arquivo fonte.
8	remove string	Remove determinada *string* (ou cadeia de *strings*) de determinada classe, função, método ou arquivo.
9	replace @annotation	Substitui determinada *annotation* de determinada classe, função ou método.

3.4.2 Instruções de customização. Esta seção detalha cada uma das nove instruções incluindo sintaxe e exemplos práticos.

[#1] **replace file**

Motivação: Necessidade de substituição de todo um artefato de software.

Sintaxe:

```
<file> replace
```

Substitui todo o artefato de software **file** pelo artefato de software com mesmo nome obtido na estrutura de arquivos auxiliares.

Exemplo prático: Aplica-se a casos onde as customizações necessárias são extensas, quando comparadas à quantidade de linhas de código do arquivo fonte customizado, e não é possível ou não é viável especificá-las com o uso das demais instruções. Por exemplo, (i) arquivos *html* com customizações extensas e de difícil representação por outras instruções e (ii) arquivos binários que não podem ser manipulados como texto, tal como imagens.

(i) `infoAluno.html replace`
(ii) `background.jpg replace`

Exemplo motivador: A Listagem 2 ilustra uma utilização da instrução de customização **replace file** no arquivo de customização da *feature* F#1:

```
1 #feature 001-leiaute
2 #description Altera o logotipo da academia e cores de
      elementos em tela
3 cb/static/grappelli/images/back/logo.jpg replace
```

Listagem 2: Fragmento do arquivo de customização F#1

Nesse exemplo, o arquivo **logo.jpg** será substituído pelo arquivo com mesmo nome que está disponível na estrutura de arquivos locais exatamente no mesmo diretório do projeto original: **cadastros_basicos/static/grappelli/images/backgrounds**.

[#2] **replace unit**

Motivação: Necessidade de substituição de toda a implementação de uma classe, função ou método no projeto customizado, i.e., desconsidera a implementação existente no projeto original.

Sintaxe:

```
<file> replace <unit'>
```

Substitui a implementação da classe, função ou método **unit'** do arquivo fonte **file**, buscando a nova implementação de **unit'** no arquivo local (aquele com mesmo nome que está disponível da estrutura de arquivos locais no mesmo diretório do projeto original).

Exemplo prático: Útil em situações onde a customização é localizada e com abrangência mais restrita, com relação ao módulo de um sistema como um todo. Por exemplo, (i) customização de uma classe responsável pela criação de objetos que renderizam um formulário, com características específicas da instituição usuária do sistema; (ii) customização de uma função que implementa uma regra de negócio que existe no projeto customizado porém diferente do projeto original e (iii) customização de um método que existe no projeto original, porém com semântica diferente no projeto customizado.

(i) `file.ext replace MatriculaForm`
(ii) `file.ext replace realizar_matricula`
(iii) `file.ext replace Matricula::validar`

Exemplo motivador: A Listagem 3 ilustra uma utilização da instrução de customização **replace unit** no arquivo de customização da *feature* F#3:

```
1 #feature 003-recepcionista-matricula
2 #description Altera a Matricula incluindo
3              recepcionista responsável
4 #dependsOn 002-recepcionista
5 alunos/models.py replace Matricula
```

Listagem 3: Fragmento do arquivo de customização F#3

De forma análoga à instrução **replace file**, a nova implementação da classe `Matricula` será recuperada no arquivo local com mesmo nome.

[#3] **replace string**

Motivação: Necessidade de substituir uma *string* (ou uma cadeia de *strings*) de uma classe, função, método ou arquivo do projeto customizado, existente no projeto original.

Sintaxe:

```
<file> replace [from <unit'>] <str'> by <str''>
```

Substitui a *string* **str'** – opcionalmente da classe, função ou método **unit'** – existente no arquivo fonte **file** pela *string* **str''**.

Exemplo prático: Instrução que se aplica a situações onde determinada *string* (ou cadeia de *strings*) de uma classe, função ou método é única no corpo dessa estrutura e necessita ser substituída por outra. Aplicável também a cenários onde é preciso substituir um termo fixo por outro, onde ambos são simples mas ocorrem mais de uma vez ao longo de um arquivo. Por exemplo, (i) substituição de uma cadeia de *strings* que representa a chamada de outro método ou função e (ii) substituição de um e-mail de contato espalhado pelo código fonte.

(i) `file.ext replace from send_alerts` *"enviar_sms()"* `by` *"enviar_email()"*
(ii) `file.ext replace` *"contato@a.br"* `by` *"contado@b.ar"*

Exemplo motivador: A Listagem 4 ilustra o uso de **replace string** no arquivo de customização da *feature* F#1:

```
cb/.../screen.css replace "#4fb2d3" by "#A10305"
cb/.../screen.css replace "#309bbf" by "#A10305"
Cb/.../contentbase.css replace "#309bbf" by "#A10305"
cb/.../rtl.css replace "#309bbf" by "#A10305"
```

Listagem 4: Fragmento do arquivo de customização F#1

Como não é informado o parâmetro opcional **unit'**, a instrução buscará por qualquer ocorrência de **str'** dentro do arquivo fonte **file**, substituindo a mesma por **str''**.

[#4] **add @annotation**

Motivação: Necessidade de adicionar uma anotação ao projeto customizado que não existe no projeto original.

Sintaxe:

```
<file> add @<annotation'> [<json>] to <unit>
       [before|after @<annotation''>]
```

Adiciona a anotação **annotation'** – opcionalmente com os atributos e valores informados no **json** – na classe, função ou método **unit** existente dentro do arquivo fonte **file**. Em caso da existência de múltiplas anotações na unidade de compilação **unit**, o mantenedor opcionalmente pode informar que deve ser adicionada antes ou após a anotação **annotation''**.

Exemplo prático: Essa instrução permite customizações simples, mas que possuem impacto na lógica do sistema customizado. Por exemplo, pode-se customizar o método **clonar_objeto** para que, diferentemente do que ocorre no projeto original, seja realizada de forma atômica no projeto customizado, como a seguir:

(único) `file.ext add @atomic to ClasseAlvo::clonar_objeto`

[#5] **add unit**

Motivação: Necessidade de adicionar uma classe, função ou método ao projeto customizado que não existe no projeto original.

Sintaxe:

```
<file> add <unit'> before|after <unit''>
```

Adiciona a classe, função ou método **unit'** ao arquivo fonte **file**. O mantenedor deve, obrigatoriamente, informar a unidade de compilação **unit''** que antecede ou sucede a unidade de compilação a ser criada (**before|after**). Esse recurso preserva a organização semântica das unidades de compilação dentro do arquivo fonte. A implementação completa da nova unidade de compilação **unit'** é obtida a partir do arquivo fonte local **file'**.

Exemplo prático: Útil em customizações em nível de módulo de um sistema, onde a criação de novas estruturas faz-se necessária. Por exemplo, (i) uma nova classe que representa uma nova entidade no sistema, (ii) uma nova função que implementa um novo *controller* em atendimento a novas regras de negócio e (iii) um novo método que implementa um relatório específico no projeto customizado.

(i) `file.ext add ServidorAfastado after Servidor`
(ii) `file.ext add print_small_details after print_details`
(iii) `file.ext add Rel::get_itens before Rel::get_material`

Exemplo motivador: A Listagem 5 ilustra uso de **add unit** no arquivo de customização da *feature* F#2:

```
#feature 002_criarModeloRecepcionista
cadastros_basicos/models.py add Recepcionista after
    OfertaDeTurma
cadastros_basicos/admin.py add RecepcionistaAdmin
    after OfertaDeTurmaAdmin
```

Listagem 5: Fragmento do arquivo de customização F#2

Assim como as instruções **replace file** e **replace unit**, a instrução **add unit** utiliza um arquivo fonte **file'** para realizar as transformações necessárias. Nesse caso, a classe, função ou método **unit'** é obtida nesse arquivo e inserida no arquivo fonte **file** antes ou após a unidade de código **unit''** já existente no mesmo.

[#6] **remove @annotation**

Motivação: Necessidade de remover uma anotação do projeto customizado, existente no projeto original.

Sintaxe:

```
<file> remove @<annotation'> from <unit>
```

Realiza a ação inversa da instrução **add_annotation** com o mesmo impacto na lógica do sistema, removendo **annotation'** da classe, função ou método **unit** existente no arquivo fonte **file**.

Exemplo prático: Customização em que um método que realiza transações no banco de dados é "atômico" no projeto original, mas que no projeto customizado é um simples método de acesso:

(único) `file.ext remove @atomic from Classe::clonar_objeto`

[#7] **remove unit**

Motivação: Necessidade de remover uma classe, uma função ou um método do projeto customizado, existente no projeto original.

Sintaxe:

```
<file> remove <unit'>
```

Remove a classe, função ou método **unit'** do arquivo fonte **file**.

Exemplo prático: Instrução utilizada em situações onde é interessante remover toda a unidade de compilação do arquivo fonte. Por exemplo, pode-se (i) remover toda uma classe desnecessária no projeto customizado, (ii) remover uma função que implementa uma ação que não pode ocorrer no projeto customizado ou (iii) remover um método sem uso.

(i) `file.ext remove ServidorLicenciado`
(ii) `file.ext remove print_full_details`
(iii) `file.ext remove EntradaPermanente::get_itens_ordenado`

[#8] **remove string**

Motivação: Necessidade de remover uma *string* (ou uma cadeia de *strings*) de uma classe, função, método ou arquivo do projeto customizado, existente no projeto original.

Sintaxe:

```
<file> remove [from <unit'>] <str'>
```

Remove a *string* **str'** – opcionalmente da classe, função ou método **unit'** – existente no arquivo fonte **file**.

Exemplo prático: Instrução criada para atender a situações onde a exclusão de uma única *string* ou de uma cadeia de *strings* representa a customização necessária para a instituição usuária do sistema.

(único) `file.ext remove from send_alerts` *"enviar_sms()"*

[#9] **`replace @annotation`**

Motivação: Necessidade de substituir uma anotação no projeto customizado que existe no projeto original.

Sintaxe:

```
<file> replace @<annotation'> [<json>] from <unit'>
```

Substitui a anotação **`annotation'`** – opcionalmente com os atributos e valores informados no **`json`** – na classe, função ou método **`unit'`** existente dentro do arquivo fonte **`file`**.

Exemplo prático: Trata-se de um "açúcar sintático" para as instruções **`remove @annotation`** e **`add @annotation`**, já que a substituição de uma anotação nada mais é que a remoção da anotação pré-existente seguida da adição da mesma com nova definição. Por exemplo, pode-se customizar o método **`get_matriculas`** para que, diferentemente do que ocorre no projeto original, seja realizada por mais de um grupo de usuários, como a seguir:

(único) `file.ext replace @group_required {"list_group": "grupo1, grupo2"} from Classe::get_matriculas`

4 FERRAMENTA DE CUSTOMIZAÇÃO

Um protótipo da ferramenta de customização denominado `ForkUpTool` foi desenvolvido para interpretar todas as instruções da linguagem de customização (Tabela 1). O protótipo foi desenvolvido em Python com uso do *framework* `Django`. A ferramenta – além de sua documentação e vídeos ilustrativos – estão publicamente disponíveis em:

https://github.com/PqES/forkuptool

A `ForkUpTool` segue uma arquitetura organizada em três módulos principais:

- *Módulo de configuração*: responsável pelas configurações, como diretório do projeto original, do projeto customizado com os arquivos locais, etc.
- *Módulo de análise de repositórios*: realiza a comparação automática de dois repositórios, destacando diferenças de forma visual e permitindo o registro de observações, quando necessário. O objetivo é avaliar se as transformações realizadas com base nas instruções de customização foram corretamente aplicadas. Além disso, extrai informações entre um *fork* de um projeto e o projeto original, destacando *commits* locais, *commits* de *merge* e a análise gráfica através de uma *timeline*. Por fim, permite a simulação de conflitos de *merge* entre um *fork* e sua origem dentro de um intervalo de datas, levantando dados como número de arquivos em conflito, número de trechos e número de linhas conflitantes;
- *Módulo de execução*: é o principal módulo do sistema que efetivamente realiza as customizações sobre a cópia do código fonte do sistema original criando portanto o sistema customizado. Possui quatro funções principais:
  - *parser de instruções de customização*: obtém as instruções de um arquivo de customização.
  - *pré-processamento*: procedimento realizado em arquivos que serão customizados. Cabeçalhos de *imports* são reescritos para melhorar sua organização, retirar duplicidades e incluir novos *imports* eventualmente necessários de acordo com as customizações.
  - *transformação de código*: responsável por manipular o código fonte do arquivo realizando a transformação especificada pela instrução. Faz uso principalmente da biblioteca `ast` da linguagem Python.
  - *orquestrador de customização*: Obtém as instruções do *parser*, as executa sequencialmente, realizando o *pré-processamento* nos arquivos alvo e realizando as devidas *transformações*. Utiliza ainda recursos adicionais como expressões regulares para busca e substituição de padrões.

5 AVALIAÇÃO

Esta seção avalia a abordagem de customização proposta em um projeto de software de grande porte frequentemente atualizado frente ao seu projeto original.

5.1 Questões de pesquisa

A avaliação proposta responde às seguintes questões de pesquisa:

QP #1: Com que frequência o projeto customizado avaliado se atualiza frente ao seu projeto original?
QP #2: A abordagem é aplicável em um cenário real?
QP #3: Existem indícios de que a abordagem evita conflitos?

5.2 Sistema alvo

O sistema real de software escolhido para avaliação foi o SUAP, apresentado na Seção 3.4.1. [3].

Um período de abrangência de quatro anos foi definido para análise, com início no momento de criação do *fork* do IFSULDEMINAS (agosto/2014) até o *merge* de atualização do mês de agosto/2018. É importante mencionar que esse período é *completamente* diferente (i.e., não inclui) o período utilizado no estudo prévio para levantamento de potenciais instruções.

No período avaliado, ocorreram 42 atualizações frente ao projeto original, ou seja, 42 mesclagens de código entre o *branch master* do sistema customizado com o *branch master* do sistema original. Essas atualizações foram identificadas com apoio do *framework* `PyDriller`[4], que permitiu buscar as integrações do tipo *merge* no repositório do projeto customizado que envolviam um ou mais *commits* do projeto original.

A rodada de avaliação #1 abrangeu o período entre 29/08/2014 e 13/11/2014, referente à data de criação do *fork* até a primeira atualização. A rodada #2 compreendeu o período entre a primeira (13/11/2014) e a segunda atualizações (19/05/2015). As demais rodadas foram estabelecidas tal como a rodada #2: período entre última atualização e a atualização seguinte.

[3] Os códigos fontes do sistema SUAP e os arquivos de customização desse sistema não podem ser publicados por questões de confidencialidade.
[4] Disponível em https://github.com/ishepard/pydriller.

5.3 Questão *QP #1*

Uma análise com foco na frequência de atualizações de um projeto customizado frente ao projeto original foi realizada de forma a assegurar que este artigo aborda um problema relevante.

5.3.1 Método. Apura-se o número de dias e o número de *commits* locais entre cada atualização. Mais importante, por meio de simulação de mesclagens de código, apura-se a quantidade de conflitos que ocorreram em cada um dos momentos de atualização.

5.3.2 Resultados e Discussão. A Tabela 2 reporta que o número de dias entre atualizações variou consideravelmente – de 0 a 202 dias – evidenciando falta de uniformidade nas atualizações do sistema do IFSULDEMINAS com relação ao sistema original. Um dos fatores que explica essa constatação é a dificuldade na resolução de conflitos de mesclagem, tal como afirmam Nishimura e Maruyama [16] e McKee et al. [12].

Após o período mais longo sem atualizações, que ocorreu na rodada #4 (202 dias) e que ocasionou 21 trechos de código com conflitos, houve uma preocupação por parte da instituição em realizar atualizações mais constantes, por exemplo, rodadas #5 à #14 ocorreram dentro de um único mês. Na rodada #15, entretanto, as atualizações frequentes deixaram de ser adotadas. Conjectura-se que os desenvolvedores envolvidos no processo de atualização tiveram dificuldades em tratar conflitos e postergaram ao máximo a atualização, que ocorreu após 88 dias, resultando em 26 conflitos. Essa é uma prática comum no desenvolvimento de sistemas, tal como evidenciado por Nicholas Nelson et al. [14].

Entre as rodadas #16 à #19 ocorreu novamente a preocupação com atualizações constantes, novamente interrompidas potencialmente pela dificuldade em se tratar conflitos de *merge*. As rodadas #20 e #21 evidenciam esse fato, com períodos mais longos entre atualizações e maior volume de conflitos. Os períodos que envolvem atualizações mais frequentes com menor volume de conflitos *versus* atualizações mais espaçadas com maior volume de conflitos se repetem ao longo de todas as rodadas.

Um fator que agrava a dificuldade em se atualizar o código fonte do projeto customizado frente ao projeto original é o maior volume de customizações próprias que tendem a aumentar o número de conflitos. Tal afirmação tem como base a análise dos períodos compreendidos entre as rodadas #32 à #34 (em destaque na Tabela 2). Esse período concentrou o maior volume percentual de *commits* realizados pelo IFSULDEMINAS (39,57% do total) e também o maior volume percentual de conflitos dentro do período analisado (44,15% do total). Foi um período longo que compreendeu 204 dias, equiparando-se àquele descrito na rodada #4.

Das rodadas #35 à #41 ocorreu novamente a preocupação em realizar atualizações constantes. O volume de *commits* próprios (média de 26,6 por rodada) foi proporcionalmente maior quando comparado ao período entre as rodadas #5 a #14 do início do projeto (média de 2,6 por rodada) quando também ocorreu essa mesma preocupação. O maior volume de *commits* locais fez com que o volume de conflitos também se elevasse de apenas dois trechos em conflitos entre as rodadas #5 e #14 para 59 trechos em conflitos entre as rodadas #35 e #41. Isso significa, de forma relativa, que elevou-se a média de 0,2 para 8,42 trechos em conflito por rodada.

Tabela 2: Períodos considerados em cada rodada de testes

		Abrangência			Commits	Conflitos	
#	Ano	Dt. inicial	Dt. final	Nº dias	Total	Arq.	Trechos
1	2014	29/08	13/11	76	3	0	0
2		13/11	19/05	187	2	2	2
3	2015	19/05	17/06	29	3	1	1
4		17/06	05/01	202	21	10	21
5		05/01	08/01	3	4	1	1
6		08/01	13/01	5	4	1	1
7		13/01	14/01	1	3	0	0
8		14/01	18/01	4	2	0	0
9		18/01	21/01	3	5	0	0
10		21/01	22/01	1	2	0	0
11		22/01	25/01	3	1	0	0
12		25/01	26/01	1	1	0	0
13		26/01	27/01	1	3	0	0
14	2016	27/01	28/01	1	1	0	0
15		28/01	25/04	88	78	13	26
16		25/04	28/04	3	6	0	0
17		28/04	29/04	1	1	2	2
18		29/04	29/04	0	1	1	1
19		29/04	02/05	3	2	0	0
20		02/05	13/07	72	4	20	43
21		13/07	16/09	65	66	20	40
22		16/09	21/09	5	6	2	8
23		21/09	29/09	8	8	3	3
24		29/09	09/01	102	100	8	12
25		09/01	11/01	2	5	0	0
26		11/01	11/01	0	1	2	2
27		11/01	02/05	111	76	33	53
28	2017	02/05	07/06	36	38	8	10
29		07/06	22/08	76	97	24	40
30		22/08	11/10	50	86	20	30
31		11/10	13/11	33	58	27	49
32		13/11	05/02	84	145	65	218
33		05/02	12/04	66	237	35	64
34		12/04	05/06	54	276	40	50
35		05/06	12/06	7	51	11	13
36		12/06	19/06	7	42	6	8
37	2018	19/06	22/06	3	28	1	1
38		22/06	29/06	7	35	8	16
39		29/06	02/07	3	4	1	6
40		02/07	04/07	2	4	3	4
41		04/07	11/07	7	22	8	11
42		11/07	08/08	28	131	7	16
		Desde o início:		**1.440**	**1.663**	**383**	**752**

5.4 Questão *QP #2*

Em uma segunda análise, avaliou-se a expressividade da linguagem de customização proposta. O objetivo é entender como a linguagem se comporta durante a construção gradativa dos arquivos de customização e se a linguagem é capaz de tratar todas as necessidades do cenário real de software avaliado.

5.4.1 Método. Os arquivos de customização são construídos de forma gradativa, rodada a rodada. O objetivo desses arquivos é manipular o código fonte do projeto original **S** de modo a transformá-lo no projeto customizado **S'**. Essa etapa foi realizada de forma manual com apoio de ferramentas de análise por um desenvolvedor do IFSULDEMINAS que é também o primeiro autor deste artigo.

Como desenvolvimento de software é um processo evolutivo, apura-se, para cada rodada, o número de arquivos criados, o número de arquivos já existentes e alterados e o número de arquivos eventualmente excluídos.

Por fim, verifica-se se o projeto customizado **S'** obtido após a aplicação das transformações descritas pelos arquivos de customização é igual ao projeto customizado resultante da resolução manual de conflitos por parte dos desenvolvedores do IFSULDEMINAS na mesclagem com o *branch master* do projeto original, em cada um dos períodos avaliados. A igualdade entre projetos é avaliada por

meio do módulo de análise de repositórios desenvolvido para esse propósito (ver Seção 4).

5.4.2 Resultados e Discussão.

Construção gradativa. Ao longo de todas as rodadas de atualização apresentadas na Tabela 2, o número de arquivos DSLs alterados foi baixo, com uma média geral de 5,56%. Em apenas 9 de 42 rodadas, os percentuais relativos ficaram acima de 10% que correspondem justamente aos períodos em que ocorreram os maiores volumes de customizações por parte do IFSULDEMINAS. Isso evidencia que boa parte das instruções de customização criadas em uma etapa anterior não necessitam de alteração pois continuam atendendo ao seu objetivo, mesmo quando o sistema original é atualizado.

Mais importante, para todas as rodadas de testes, o projeto customizado obtido a partir das instruções de customização foi igual ao projeto customizado resultante da resolução manual de conflitos por parte dos desenvolvedores do IFSULDEMINAS na mesclagem com o *branch master* do projeto original. Ressalta-se que, em apenas quatro dos 195 arquivos customizados, a ferramenta proposta falhou ao tratar uma instrução particular do *framework* `Django` relacionada ao registro de *models* no ambiente de administração automática desse *framework*. A falha gerou erro de execução da ferramenta e a correção para esse problema será contemplada em uma versão futura do protótipo.

Expressividade da linguagem. A Tabela 3 descreve o uso de instruções de customização. As nove instruções de customização propostas mostraram-se eficazes em realizar as customizações de um projeto durante quatro anos.

Tabela 3: Uso de instruções de customização

Uso por instrução						
Instrução		**Nº utiliz.**		**% utiliz.**		**Nº dsls**
replace file		122		23,69%		122
replace string		44		8,54%		27
	classe		*74*		*30,20%*	
replace unit	*função*	245	*93*	47,57%	*37,96%*	58
	método		*78*		*31,84%*	
add @annotation		6		1,17%		2
	classe		*28*		*30,77%*	
add unit	*função*	91	*53*	17,67%	*58,24%*	36
	método		*10*		*10,99%*	
remove @annotation		2		0,39%		2
remove string		0		0,00%		0
remove unit		0		0,00%		0
replace @annotation		5		0,97%		2

Embora especificadas nove instruções de customização, notou-se o uso expressivo de três instruções em particular. A primeira delas – **`replace unit`** – foi utilizada 245 vezes ao longo de 58 arquivos de customização distintos (47,57% do total). Apesar do amplo uso, percebe-se pela análise da Tabela 3 que houve uma distribuição relativamente uniforme dessa utilização com relação ao tipo de unidade de código tratada: funções (37,96% do total), métodos (31,84%) e classes (30,20%). Essa distribuição segue, portanto, uma linha onde customizações mais específicas (funções e métodos) foram tratadas antes de customizações mais amplas (classes).

Acredita-se que a ampla utilização de **`replace unit`** se deu pelo fato de a instrução representar uma customização que se adequa à maioria das necessidades de customização, possibilitando a tratativa de situações em diferentes níveis de granularidade.

A segunda instrução mais utilizada – **`replace file`** – teve seu uso expressivo justificado pela forma como a abordagem de customização foi idealizada, aliada às características do projeto customizado. Como se trata de um projeto web, o uso de arquivos HTML (*HyperText Markup Language*), CSS (*Cascading Style Sheets*) e JavaScript é significativo. No entanto, as instruções de customização focam na linguagem principal do sistema e os arquivos supracitados são, em geral, customizados "por completo" substituindo-se o arquivo fonte **`file`** pelo arquivo fonte **`file'`** correspondente.

A terceira instrução mais utilizada – **`add unit`** – teve seu uso justificado por sua semântica: criação de novas classes, funções e métodos. Isso permitiu principalmente customizações que agregaram novas *features* ao sistema, parte significativa do processo como um todo.

Em uma última análise, observou-se a ausência de utilização das instruções **`remove unit`** e **`remove string`**. Embora idealizadas e implementadas durante a prova de conceito descrita na Seção 3.4.1, não foi observada a necessidade de sua utilização ao longo das 42 rodadas de avaliação.

5.5 Questão *QP #3*

Em uma terceira análise, observou-se se a abordagem proposta contribui na diminuição dos conflitos de mesclagem.

5.5.1 Método. Analisa-se números de conflitos que ocorreram ao longo das 42 atualizações de código e o número de arquivos de customização criados e mantidos ao longo desse mesmo período.

5.5.2 Resultados e Discussão. Durante as atualizações frente ao projeto original usando a abordagem proposta, não houve ocorrência de conflitos de mesclagem, já que a abordagem sempre busca o código fonte do projeto original e aplica sobre ele as transformações necessárias; não há mesclagem de código e, sendo assim, a ausência de conflitos se justifica.

Conforme apresentado na Tabela 2, conflitos em 752 trechos de código deixaram de ocorrer. Com a abordagem proposta, evitou-se uma média de ~18 trechos em conflito por atualização.

Por fim, observou-se que quanto mais customizado é o sistema mais conflitos de mesclagem ocorrem, independentemente de quão frequentes ou não são realizadas as atualizações. Em uma análise Pearson da correlação pareada entre número de *commits* locais e número de conflitos observou-se alta correlação dos valores apresentados ($rho = 0.7786$).

5.6 Discussão

A *QP #1* demonstrou que a frequência de atualização do projeto customizado frente ao *branch master* do projeto original foi relativamente alta. Ao longo dos 48 meses analisados, ocorreram 42 atualizações, o que implica aproximadamente em uma atualização a cada um mês e cinco dias. Demonstrou ainda que uma frequência curta ou longa de atualizações não soluciona a ocorrência de conflitos de *merge*, i.e., os conflitos persistiram mesmo com atualizações constantes de código.

A *QP #2* demonstrou que, ao longo de um período de quatro anos, arquivos de customização se mostraram suficientes para se obter o

projeto customizado **S'** a partir do código fonte do projeto original. Em resumo, **S'** foi sempre equivalente ao sistema customizado pelos próprios desenvolvedores do IFSULDEMINAS. Ademais, o número de arquivos de customização cresceu de forma gradativa e com baixos percentuais de alteração.

Por fim, a *QP #3* mostrou que a abordagem evita conflitos em cenários reais. Com a abordagem proposta, conflitos em 752 trechos de código deixaram de ocorrer. Isso significa que menos tempo e trabalho poderiam ter sido empregados para manter o sistema atualizado, ajudando a equipe de desenvolvimento a cumprir prazos e atingir metas.

5.7 Limitações

Embora tenha-se uma avaliação com resultados bem positivos, é importante destacar que **não** há garantias diretas de qualidade. A adoção da abordagem não dispensa a realização de outras atividades comuns em cenários de resolução de conflitos, como a realização de inspeções e testes. Logo, as garantias de qualidade que a abordagem oferece são as mesmas que a resolução manual de conflitos, isto é, podem sim ocorrer problemas que precisam ser detectados por outras atividades do processo de desenvolvimento de software.

Este estudo **não** foca em avaliar a questão de esforço. Aceita-se que o esforço de desenvolvimento pode ser até maior a curto prazo, mas o fato de modularizar – em 1 arquivo – *n* modificações que produzem uma nova *feature* ou alterem uma existente facilita a manutenibilidade a longo prazo, além de evitar um código confuso que mistura customizações com o código original.

O objetivo da abordagem **não** é evitar conflitos de *merge*, mas prover uma forma não *ad hoc* de conduzir projetos que são sempre atualizados frente a um projeto original onde *features* são modularizadas, documentadas, rastreáveis e podem ser reutilizadas. Logo, sua adoção pode sim gerar alguns tipos de *conflitos de customização* que devem ser considerados quando a mesma é aplicada:

- *Customização em arquivo modificado na origem **com** impacto na execução da ferramenta.* Assuma, por exemplo, uma customização que substitui um método `foo` do projeto original. Caso esse método seja excluído no projeto original, a abordagem falhará alertando o ocorrido em contrapartida a um *merge* que poderia encobrir tal mudança sem gerar sequer conflito.
- *Customização em arquivo modificado na origem **sem** impacto na execução da ferramenta.* Assuma, por exemplo, uma customização que remove uma certa *string* do projeto original. Todas as novas *strings* adicionadas no projeto original serão removidas por tal customização, o que nem sempre pode ser o desejado. Por isso, optar por granularidade fina e realizar testes funcionais são fundamentais.
- *Customização em arquivo excluído na origem.* Similarmente ao 1^o item, a abordagem falhará alertando o ocorrido em contrapartida a um *merge* que poderia encobrir tal mudança sem gerar sequer conflito.

5.8 Ameaças à validade

É possível identificar duas ameaças à validade para este estudo.

Validade interna: A criação dos arquivos de customização foi realizada pelo primeiro autor deste artigo e, embora não houve revisão por pares, o autor deste artigo é também desenvolvedor do IFSULDEMINAS e apenas utilizou as instruções propostas para a customização de trechos de código do projeto original como havia sido feito após resolução de conflitos de *merge*.

Validade externa: Embora a expressividade tenha demonstrado eficaz, não se pode afirmar que a abordagem proposta fornecerá resultados equivalentes em outros sistemas, o que é usual em estudos de caso em engenharia de software. No entanto, foi analisado um período longo de quatro anos em um projeto real de grande porte.

6 TRABALHOS RELACIONADOS

Os trabalhos relacionados foram divididos em três grupos: (i) recursos do `Git`, que procuram auxiliar no processo de mesclagem e resolução de conflitos e podem ser confundidos com a abordagem proposta neste artigo; (ii) previsão ou antecipação de conflitos, que tentam antecipar ações para evitar a ocorrência de conflitos ou evitar que se tornem mais difíceis de resolver; e (iii) apoio na resolução de conflitos, que encaram os conflitos de mesclagem como inevitáveis e buscam apoiar a resolução dos mesmos através de diferentes abordagens.

Recursos do Git: A opção *ours* disponível no *git merge* – que sinaliza ao Git que em conflitos deve-se optar pelo lado *ours* do par de arquivos conflitantes – assemelha-se à abordagem proposta neste artigo. No entanto, cabe ressaltar algumas vantagens da abordagem proposta: (i) não se aplica exclusivamente a repositórios Git; (ii) "*ours*" força a escolha de um "lado" no momento do *merge* (*ours* ou *theirs*) para **todo** o *merge* realizado, enquanto que a abordagem proposta permite especificar pela implementação local (*ours*) em diferentes níveis de granularidade, e.g., para um método ou um trecho específico; e (iii) a abordagem proposta não propõe evitar apenas a ocorrência de conflitos, mas promover modularidade, documentação, rastreabilidade e reúso.

O *git rerere* [4], acrônimo de *reuse recorded resolution* ("reutilizar resolução gravada"), permite ao `Git` buscar no histórico de mesclagens bem sucedidas soluções de conflitos para situações semelhantes. Apesar de benéfico, é um recurso com escopo limitado e sua semântica é completamente diferente da semântica da abordagem de customização proposta neste artigo.

O *git patch* [4], por sua vez, é um recurso que permite a geração de "arquivos de *patch*" que nada mais são que representações das alterações realizadas por um *commit* específico. Portanto, sua semântica difere da semântica dos arquivos de customização deste projeto. Além disso, a aplicação de um arquivo de *patch* do `Git` pode resultar nos mesmos conflitos de uma mesclagem entre *branches* realizada com `git merge`.

Previsão ou antecipação de conflitos. Guimarães e Silva [6] afirmam que a detecção antecipada de conflitos permite aos desenvolvedores atuar de forma pró-ativa sobre o código fonte. Portanto, os autores propõem uma solução e implementam um *plug-in* para o IDE Eclipse que mescla continuamente os trabalhos ainda não confirmados em um sistema em segundo plano de forma a detectar os conflitos de forma antecipada, o que denominam de "mesclagem contínua".

Similarmente, Kasi e Sarma [8] apresentam uma nova técnica de minimização de conflitos em que agrupa-se as mudanças previstas para um projeto em tarefas, avalia-se as restrições de cada

tarefa e recomenda-se uma ordem ótima de realização de tarefas para cada desenvolvedor. Os autores desenvolveram um sistema denominado `Cassandra` que permite o agendamento das tarefas tratando conflitos em nível de tarefa e não de código.

Os estudos deste grupo [6, 8] não se aplicam ao cenário de atualização de *forks* apresentado neste artigo. A mesclagem de código, nesse cenário, envolve trabalhos já persistidos em *branches* de produção pertencentes a projetos distintos, ou seja, códigos já finalizados. Os estudos apresentados abordam estratégias para códigos em desenvolvimento, pertencentes a um mesmo projeto.

Apoio na resolução de conflitos. Nishimura e Maruyama [16] propõem o `MergeHelper`, uma ferramenta que explora o histórico de operações de edição do código-fonte para apoiar a resolução de conflitos de mesclagem. A ferramenta provê funções dentro do `IDE Eclipse` de forma a facilitar a tarefa de resolução de conflitos de mesclagem, explorando o histórico de edições que ela gera e guarda ao longo do tempo. Esse trabalho não se aplica ao cenário proposto neste artigo, pois o histórico de operações de edição não existe no cenário de mesclagem de *forks*, embora possa ser adaptado.

Similarmente, Costa et al. [5] propõem o `TIPMerge`, uma ferramenta que identifica e recomenda os desenvolvedores mais aptos a atuar na resolução de conflitos de mesclagem, com base em sua experiência passada dentro do projeto, nas alterações por eles realizadas e nas dependências com relação aos arquivos envolvidos. Esse trabalho também não se aplica ao cenário proposto neste artigo, pois, mesmo que um histórico de desenvolvedores fosse construído, a estratégia potencialmente indicaria um desenvolvedor da equipe do projeto de origem para resolução de um conflito e não do projeto customizado.

7 CONCLUSÃO

O *problema* abordado neste artigo é manter sistemas customizados atualizados frente aos projetos originais. A mesclagem do código do projeto original no projeto customizado pode gerar inúmeros conflitos por se tratar de dois projetos independentes com diferentes equipes trabalhando de forma paralela e, mais relevante, a equipe do projeto original sequer tem conhecimento do que vem sendo customizado em seus *forks*.

Diante disso, este artigo propôs, implementou e avaliou uma abordagem para atualização de *forks* frente ao projeto original que evitou 752 conflitos que ocorreram durante quatro anos em um projeto customizado de grande porte. A ideia central é que o projeto customizado seja desenvolvido como um "conjunto de alterações que devem ser aplicadas no projeto original" o que promove:

(i) *redução de conflitos*: o código do projeto original não é mais alterado diretamente;
(ii) *modularidade*: customizações de uma *feature* encontram-se especificadas em um único arquivo;
(iii) *documentação*: informações relevantes de cada *feature* estão nos arquivos de customização, assim como as dependências;
(iv) *rastreabilidade*: é possível rastrear a *feature* a qual um trecho de código pertence; e
(v) *reúso*: basta fornecer o arquivo de customização a um outro projeto customizado.

Trabalhos futuros incluem (i) a avaliação da abordagem em um projeto customizado desde o início do desenvolvimento; (ii) criação de instruções de granularidade fina para arquivos JavaScript, CSS, HTML, etc.; e (iii) a definição de um conjunto de indicadores que auxiliem na recomendação de refatorações em arquivos de customização existentes, por exemplo, existe um **`replace class`** embora apenas um método é de fato alterado.

AGRADECIMENTOS

Esta pesquisa é apoiada pelo IFSULDEMINAS, FAPEMIG (APQ-03513-18) e CNPq (305829/2018-1).

REFERÊNCIAS

[1] F. F. Blauw. 2018. The Use of Git as Version Control in the South African Software Engineering Classroom. In *13th IST-Africa Week Conference (IST-Africa)*. 1–8.
[2] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. In *32rd IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 334–344.
[3] Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. 2015. Assessing semistructured merge in version control systems: A replicated experiment. In *9th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–10.
[4] Scott Chacon and Ben Straub. 2014. *Pro git.* Apress.
[5] Catarina Costa, Jair Figueiredo, Leonardo Murta, and Anita Sarma. 2016. TIPMerge: recommending experts for integrating changes across branches. In *24th International Symposium on Foundations of Software Engineering (ESE/FSE)*. 523–534.
[6] Mário Luís Guimarães and António Rito Silva. 2012. Improving early detection of software merge conflicts. In *34th International Conference on Software Engineering (ICSE)*. 342–352.
[7] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. 2017. Why and how developers fork what from whom in GitHub. *Empirical Software Engineering* 22, 1 (2017), 547–578.
[8] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In *35th International Conference on Software Engineering (ICSE)*. 732–741.
[9] Olaf Leßenich, Sven Apel, Christian Kästner, Georg Seibt, and Janet Siegmund. 2017. Renaming and shifted code in structured merging: Looking ahead for precision and performance. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 543–553.
[10] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2017. FHistorian: Locating features in version histories. In *21st International Systems and Software Product Line Conference (SPLC)*. 49–58.
[11] Wardah Mahmood, Moses Chagama, Thorsten Berger, and Regina Hebig. 2020. Causes of merge conflicts: a case study of ElasticSearch. In *14th International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS)*. 1–9.
[12] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. 2017. Software practitioner perspectives on merge conflicts and resolutions. In *33rd IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 467–478.
[13] Tom Mens. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 28, 5 (2002), 449–462.
[14] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. 2019. The life-cycle of merge conflicts: processes, barriers, and strategies. *Empirical Software Engineering* 24, 1 (2019), 1–44.
[15] Hoai Le Nguyen and Claudia-Lavinia Ignat. 2018. An Analysis of Merge Conflicts and Resolutions in Git-Based Open Source Projects. *Computer Supported Cooperative Work* 27, 3 (2018), 741–765.
[16] Yuichi Nishimura and Katsuhisa Maruyama. 2016. Supporting merge conflict resolution by using fine-grained code change history. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 661–664.
[17] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. 2012. Managing forked product variants. In *16th International Software Product Line Conference (SPLC)*. 156–160.
[18] D. Spinellis. 2012. Git. *IEEE Software* 29, 3 (2012), 100–101.
[19] Ștefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and integrated variants in an open-source firmware project. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 151–160.
[20] Shurui Zhou, Stefan Stanciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wasowski, and Christian Kästner. 2018. Identifying features in forks. In *40th International Conference on Software Engineering (ICSE)*. 105–116.