

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXTAS E DA TERRA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

**AVALIAÇÃO SISTEMÁTICA DE UMA
ABORDAGEM PARA INTEGRAÇÃO DE
FUNCIONALIDADES EM SISTEMAS
WEB CLONADOS**

Por
JADSON JOSÉ DOS SANTOS
Dissertação de Mestrado

NATAL

Agosto, 2015

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

JADSON JOSÉ DOS SANTOS

**Avaliação Sistemática de uma
Abordagem para Integração de
Funcionalidades em Sistemas Web
Clonados**

Dissertação submetida à Coordenação do Programa de Pós-Graduação em Sistemas e Computação, do Centro Ciências Exatas e da Terra, da Universidade Federal do Rio Grande do Norte, como parte dos requisitos para obtenção de título de Mestre em Sistemas e Computação.

Orientador: Uirá Kulesza, Dr.

NATAL

Agosto, 2015

UFRN / Biblioteca Central Zila Mamede
Catalogação da Publicação na Fonte

Santos, Jadson José dos

Avaliação sistemática de uma abordagem para integração de
funcionalidades em sistemas web clonados. / Jadson José dos Santos. –
Natal, RN, 2015.

113 f. : il.

Orientador: Prof. Dr. Uirá Kulesza.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do
Norte. Centro de Ciências Exatas e da Terra. Programa de Pós-
Graduação em Sistemas e Computação.

1. Linhas de produtos de software – Dissertação. 2. Clonagem de
sistemas web - Dissertação. 3. Análise de conflitos de Merge -
Dissertação. 4. Merge de código fonte – Dissertação. I. Kulesza, Uirá.
III. Universidade Federal do Rio Grande do Norte. IV. Título.

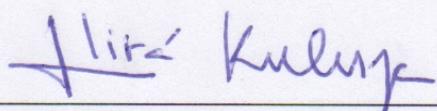
RN/UF/BCZM

CDU 004.4

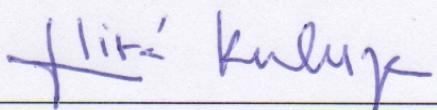
JADSON JOSÉ DOS SANTOS

Avaliação Sistemática de uma Abordagem para Integração de Funcionalidades em Sistemas Web Clonados

Esta Dissertação foi julgada adequada para a obtenção do título de mestre em Sistemas e Computação e aprovado em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

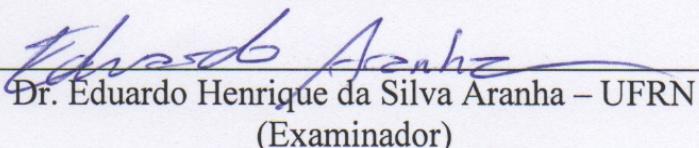


Dr. Uirá Kulesza – UFRN
(Presidente)

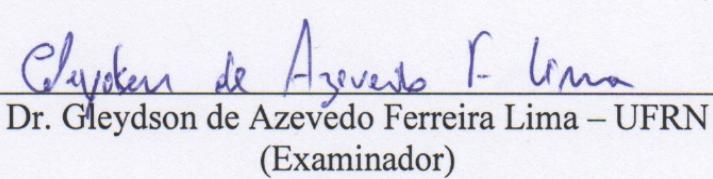


Dr. Uirá Kulesza – UFRN
(Coordenador do Programa)

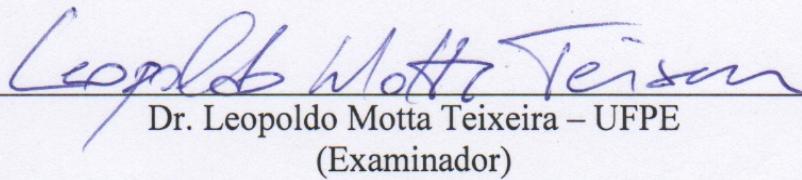
Banca Examinadora



Dr. Eduardo Henrique da Silva Aranha – UFRN
(Examinador)



Dr. Gleydson de Azevedo Ferreira Lima – UFRN
(Examinador)



Dr. Leopoldo Motta Teixeira – UFPE
(Examinador)

Agradecimentos

Primeiramente agradeço ao Prof. Dr. Uirá Kulesza por ter me aceitado como orientando, pelas horas gastas com reuniões, escrita e revisão de artigos (muitas vezes durante as férias ou finais de semana) e pela paciência de esperar até os 45 minutos do segundo tempo para eu conseguir concluir esse trabalho.

Agradeço às pessoas que contribuíram com esse trabalho. Agradeço à Gleydson Lima por ter cedido os dados da evolução do sistema *Target* utilizado nesse estudo. Agradeço à Daniel Alencar que contribuiu com o módulo de mineração das evoluções do código fonte da ferramenta. E agradeço à Fladson Gomes que contribuiu no suporte ao Git e GitHub existente na ferramenta.

Agradeço aos professores que aceitaram fazer parte da minha banca de defesa, Prof. Dr. Eduardo Aranha, Prof. Dr. Leopoldo Teixeira e Dr. Gleydson Lima.

Agradeço à Superintendência de Informática da UFRN por ceder o código fonte do SIGAA para realização desse estudo e pela liberação durante o expediente para assistir às aulas ou participar de reuniões do mestrado.

Agradeço à minha agora esposa Luanna Rocha, no começo dessa jornada ainda éramos apenas namorados, que tentou não me ocupar nos finais de semana para que eu pudesse me dedicar ao mestrado e que recentemente me esperou vários dias sozinha até as 22 horas, em casa, enquanto eu tentava evoluir esse trabalho.

Agradeço ao meu irmão, porque eu sei que, se eu não falar nele aqui, ele vai ficar com raiva.

Agradeço à minha Mãe que sempre cuidou de mim.

Por fim, dedico este trabalho ao meu pai, José Vital dos Santos, por todo o esforço que sempre fez, trabalhando dia e noite para conseguir me manter e pagar meus estudos. Pai, o senhor pode dizer que conseguiu formar um filho mestre. Muito obrigado.

*Se você quer ser bem sucedido, precisa ter dedicação total, buscar seu
último limite e dar o melhor de si.*

Ayrton Senna.

Resumo

A engenharia de linhas de produto de software traz vantagens quando comparado ao desenvolvimento tradicional de sistemas no que diz respeito a customização em massa dos seus componentes, reduzindo o custo e aumentando a qualidade dos produtos de uma família de sistemas. Contudo, em determinados cenários, a manutenção de cópias separadas – clones – de um sistema tem sido explorada como uma abordagem para gerência de variabilidades, por ser mais simples e fácil de gerenciar. Esta dissertação de mestrado busca avaliar qualitativamente uma abordagem proposta para auxiliar a reconciliação de funcionalidades entre sistemas que foram clonados. A abordagem analisada é baseada na mineração de informações de evoluções dos sistemas contemplando uma análise de tipos específicos de conflitos que tem por finalidade indicar possíveis problemas na integração de funcionalidades entre versões clonadas de um mesmo sistema, não indicados por ferramentas tradicionais de controle de versão. Os resultados do estudo mostram a viabilidade de utilização da abordagem, dentro do cenário analisado, além de caracterizar os tipos e a complexidade de conflitos na integração de um sistema web de larga escala.

Palavras-chave: Linhas de Produtos de Software, Clonagem de Sistemas Web, Análise de Conflitos de Merge, Merge de Código Fonte.

Abstract

The software product line engineering brings advantages when compared with the traditional software development regarding the mass customization of the system components. However, there are scenarios that to maintain separated clones of a software system seems to be an easier and more flexible approach to manage their variabilities of a software product line. This dissertation evaluates qualitatively an approach that aims to support the reconciliation of functionalities between cloned systems. The analyzed approach is based on mining data about the issues and source code of evolved cloned web systems. The next step is to process the merge conflicts collected by the approach and not indicated by traditional control version systems to identify potential integration problems from the cloned software systems. The results of the study show the feasibility of the approach to perform a systematic characterization and analysis of merge conflicts for large-scale web-based systems.

Keywords: Software Product Lines, Cloned Web Systems, Merge Conflict Analysis, Source Code Merge.

Sumário

1. INTRODUÇÃO	1
1.1. Contextualização	1
1.2. Problema	3
1.3. Limitações dos Trabalhos Atuais	4
1.4. Objetivos.....	5
1.5. Questões de Pesquisa.....	6
1.6. Organização do Documento	6
2. FUNDAMENTAÇÃO TEÓRICA.....	8
2.1. Linhas de Produtos de Software	8
2.1.1 Similaridades e Variabilidades	9
2.1.2 Features	10
2.1.3 Processo de Derivação do Produto	11
2.1.4 Técnicas de Implementação de Variabilidades	12
2.1.5 Fases da Engenharia de Linhas de Produtos	13
2.2. Gerenciamento de Configuração e Mudanças.....	15
2.3. Mineração de Repositórios	17
2.4. Merge de Código Fonte.....	17
2.4.1 Two-Way Merge	18
2.4.2 Three-Way Merge	18
3. ABORDAGEM DE MERGE PARA SISTEMAS CLONADOS	20
3.1. Visão Geral.....	20
3.2. Módulo de Mineração das Evoluções	24
3.3. Módulo de Análise de Conflitos	28
3.3.1 Conflitos Diretos	28
3.3.2 Conflitos Indiretos	29
3.3.3 Pseudo Conflitos.....	31
3.4. Estratégia de Resolução de Conflitos.....	32

3.5. Evoluções Técnicas Realizadas na Abordagem	36
3.5.1 Evoluções Técnica no Cálculo dos Conflitos Indiretos.....	36
3.5.2 Análise de Dependência entre Tarefas	41
4. ESTUDO EMPÍRICO.....	43
4.1. Objetivos e Questões de Pesquisa.....	43
4.2. Seleção do Sistema para o Estudo.....	45
4.3. Metodologia do Estudo	46
4.4. Resultados do Estudo Empírico	51
4.4.1 QP1: Tarefas identificadas pela abordagem como não tendo conflitos podem de fato ser integradas?	51
4.4.2 QP2: Tarefas indicadas com conflitos diretos, realmente representam problemas de integração?	56
4.4.3 QP3: Os conflitos indiretos identificados pela abordagem representam conflitos que trazem problemas para a integração?	64
4.5. Limitações e Ameaças ao Estudo	75
4.5.1 Ameaças Internas:	75
4.5.2 Ameaças Externas:	76
5. TRABALHOS RELACIONADOS	77
5.1. Trabalhos relacionados à Resolução de Conflitos	77
5.2. Trabalhos relacionados à Clonagem de Sistemas.....	80
6. CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS.....	83
6.1. Análise das Questões de Pesquisa da Dissertação	83
6.2. Revisão das Contribuições.....	84
6.3. Limitações do Trabalho	86
6.4. Trabalhos Futuros	87
REFERÊNCIAS BIBLIOGRÁFICAS	88

Índice de Figuras

<i>Figura 1-1: Adição da Feature Chat On-line da Turma Virtual</i>	3
<i>Figura 1-2: Reconciliação de funcionalidades entre clones.....</i>	3
<i>Figura 2-1: Economia no uso de Linha de Produtos de Software.....</i>	9
<i>Figura 2-2: Representações dos tipos de Features usando a representação FODA.....</i>	11
<i>Figura 2-3: As técnicas básicas de realizar variabilidades em Linhas de Produtos de Software.....</i>	12
<i>Figura 2-4: Ciclos de vida da Engenharia de Linhas de Produtos de Software</i>	14
<i>Figura 2-5: Exemplo de Conflito de Edição</i>	16
<i>Figura 2-6: Two-Way Merge</i>	18
<i>Figura 2-7: Three-Way Merge</i>	19
<i>Figura 3-1: Visão Geral do Processo de Merge entre sistemas web clonados</i>	21
<i>Figura 3-2: Visão Geral da Arquitetura do MergeClear.....</i>	23
<i>Figura 3-3: Diagrama de Classes do MergeClear</i>	24
<i>Figura 3-4: Modelo ChangeLogHistory</i>	27
<i>Figura 3-5: Exemplo de um Conflito Direto</i>	29
<i>Figura 3-6: Representação de um Conflito Indireto.....</i>	30
<i>Figura 3-7: Exemplo de Pseudo Conflito.....</i>	31
<i>Figura 3-8: Estratégia de Merge com base no tipo de Conflito</i>	34
<i>Figura 3-9: Conflitos detectados pelo MergeClear</i>	35
<i>Figura 3-10: Visualização da Árvore e Conflitos da Evolução</i>	36
<i>Figura 3-11: Grafo de chamadas usado na versão anterior da ferramenta.....</i>	37
<i>Figura 3-12: Grafo de Chamadas completo entre artefatos usados na análise de conflitos indiretos</i>	38
<i>Figura 3-13: Assinatura que identifica unicamente um artfato.....</i>	40
<i>Figura 3-14: Referências em Comentários Retornadas pelo JDT.....</i>	40
<i>Figura 3-15: Árvore de Dependência entre Tarefas.....</i>	41
<i>Figura 4-1: Evoluções a partir do ponto de clonagem</i>	44
<i>Figura 4-2: Configurações da Ferramenta de Merge utilizadas para o estudo</i>	48
<i>Figura 4-3: Dependências entre Tarefas</i>	52
<i>Figura 4-4: Workspace onde foi realizado a integração</i>	56
<i>Figura 4-5: Classes alteradas durante a integração manual para responder a QPI</i>	56

<i>Figura 4-6: Mudança da tarefa 87467 no Source</i>	58
<i>Figura 4-7: Mudança da tarefa 87467 no Target.....</i>	59
<i>Figura 4-8: Adição de mais um papel para realizar o login no sistema</i>	60
<i>Figura 4-9: Alteração realizada no Target.....</i>	60
<i>Figura 4-10: Alterações do método processar no Source</i>	61
<i>Figura 4-11: Alterações do método processar no Target.....</i>	62
<i>Figura 4-12: Correção do erro da consolidação no lado do Source</i>	63
<i>Figura 4-13: Correção do erro da consolidação no lado do Target.....</i>	63
<i>Figura 4-14: Análise do Grafo de Chamadas para a tarefa 61549.....</i>	67
<i>Figura 4-15: Extração de código fonte para um método.....</i>	69
<i>Figura 4-16: Alteração do Método TurmaMbean#atualizar()</i>	69
<i>Figura 4-17: Alteração do método TurmaGraduacaoMBean#isDefineDocentes()</i>	70
<i>Figura 4-18: Alteração Regras de Validação no Processador.....</i>	71
<i>Figura 4-19: Adição de um campo na busca no sistema Source</i>	71
<i>Figura 4-20: Customização do sistema Target, que é afetada indiretamente pela mudança do Source.</i>	72
<i>Figura 4-21: Busca JDT para o método abstrato toString()</i>	73
<i>Figura 4-22: Artefatos ignorados durante a análise desse trabalho.....</i>	74

Índice de Tabelas

<i>Tabela 4-1: Tamanho e quantidade de usuário do SIGAA</i>	45
<i>Tabela 4-2: Resultado da Análise de Conflitos da Evolução Estudada.....</i>	47
<i>Tabela 4-3: Classificação das Tarefas da Evolução do Sistema Source</i>	49
<i>Tabela 4-4: Classificação das Tarefas da Evolução do Sistema Target</i>	49
<i>Tabela 4-5: Médias de classes Java alteradas por tipo de tarefa e tipo de conflito no Source.....</i>	50
<i>Tabela 4-6: Média de classes Java alteradas por tipo de tarefas e tipo de conflito no Target</i>	50
<i>Tabela 4-7: Quantitativo das tarefas analisadas</i>	53
<i>Tabela 4-8: Quantitativo das dependências entre tarefas</i>	54
<i>Tabela 4-9: Análise dos Conflitos diretos.....</i>	57
<i>Tabela 4-10: Conflitos Indiretos Detectados por Nível de Profundidade da Análise</i>	64
<i>Tabela 4-11: Análise dos conflitos indiretos apresentados pela ferramenta.....</i>	66

1. Introdução

1.1. Contextualização

Linha de Produtos de Software (LPS) [Clements, P., and L. Northrop. 2001] surgiu como uma nova abordagem de desenvolvimento de sistemas de software que visa permitir um reuso sistemático de seus componentes, em torno do negócio da empresa para a qual ela está sendo aplicada. Com a criação de variações para funcionalidades específicas implementadas pelo sistema é possível configurá-lo para executar em vários cenários. Assim, essa abordagem tem como objetivo reduzir o custo de desenvolvimento e aumentar a qualidade dos sistemas, a partir do reuso e customização em massa [Krueger 2001] de uma família de sistemas, para um determinado segmento do mercado.

Embora a abordagem de LPS traga vantagens quando comparado ao desenvolvimento tradicional de sistemas de software, o desenvolvimento, manutenção e evolução de uma LPS pode ser desafiador em determinados cenários. Entre os cenários desafiadores para implantar a abordagem de linhas de produtos encontram-se: (i) não dispor do tempo necessário para desenvolver e evoluir gradualmente o núcleo e pontos de variações da LPS; (ii) alto investimento inicial requerido; (iii) distribuição geográfica e/ou falta de capacidade técnica da equipe de desenvolvimento para customizar a LPS para um número grande de clientes; (iv) em cenários onde os produtos apresentam certa similaridade em determinadas partes ou módulos, mas em outras partes, há grandes diferenças entre variações, que dificultam a aplicação da estratégia de linhas de produtos em toda a extensão do código fonte do sistema; ou ainda (v) uma LPS tem seu núcleo e variabilidades alterados por diferentes e independentes equipes de desenvolvimento, sem que exista uma equipe que possa coordenar e monitorar sua evolução.

Considerando os problemas citados, muitas empresas de desenvolvimento de software [Dubinsky Y., et al. 2013] optam por realizar um *fork* no código dos seus sistemas quando elas têm que atender a cenários que apresentam variações nas suas regras de negócio. Esse *fork* gera sistemas que são inicialmente clones um do outro e que passam a evoluir de forma paralela ao longo do tempo [Rubin et al. 2012] [Rubin, Czarnecki and Chechik 2013].

Mesmo com os produtos clonados evoluindo de forma não organizada, várias características permanecem comuns entre eles. Também uma nova característica (*feature*) acrescentada em um produto pode ser de interesse de outro. Considere, por exemplo, o caso do sistema SIGAA – Sistema Integrado de Gestão de Atividades Acadêmica [SIGAA, 2014]. O SIGAA é um sistema de informação acadêmico desenvolvido e mantido pela Universidade Federal do Rio Grande do Norte, que gerencia a maioria das atividades acadêmicas dessa instituição, nos diferentes níveis de ensino, tais como infantil, ensino, graduação, pós-graduação, pesquisa, extensão, entre outros. Esse sistema possui diversas variações nas suas regras de negócio que foram adicionadas para atender as demais instituições que o utilizam. Contudo, essas variações nas regras de negócio não foram suficientes para atender a determinados módulos. A medida que o sistema foi sendo implantado em outras instituições, as demandas pela criação de novas variabilidades em regras de negócio cresceram vertiginosamente.

Quando o SIGAA foi disponibilizado [Sena, D., et al. 2012] [Santos, J., et al. 2012] para outras instituições de ensino superior brasileiras, o ambiente virtual de aprendizado, denominado “Turma Virtual”, por exemplo, não possuía a funcionalidade de “Chat On-line” como um de seus módulos. Quando essa funcionalidade foi disponibilizada, várias outras instituições que utilizam o SIGAA, se interessaram em adicioná-la aos seus produtos. Contudo, ao tentar copiar os artefatos de código que implementavam a funcionalidade “Chat On-line” para as outras versões clonadas do sistema, percebeu-se que esse tipo de processo gera vários conflitos para integração do código fonte das diferentes versões clonadas e evoluídas do sistema. A Figura 1-1 mostra a evolução do modelo de *features* do SIGAA, com a adição da *feature* “Chat On-line” que se deseja repassar aos seus sistemas clones.

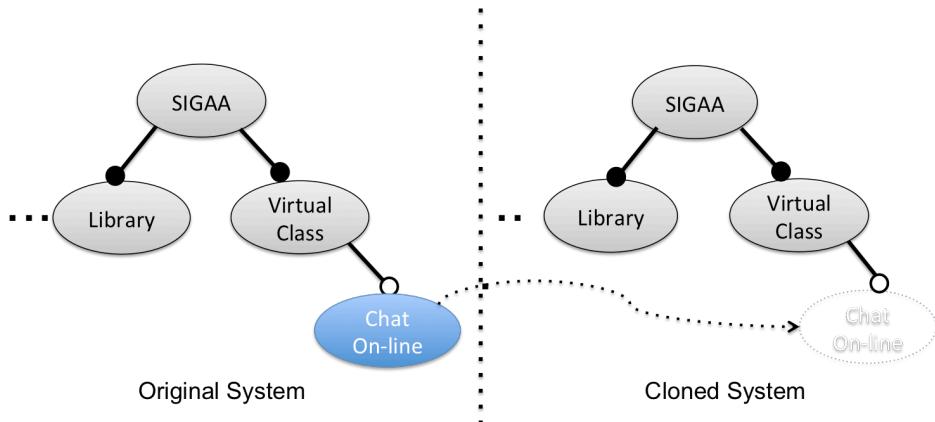


Figura 1-1: Adição da Feature Chat On-line da Turma Virtual

1.2. Problema

Neste contexto, foi proposta uma abordagem para tentar minimizar o problema de reconciliação de funcionalidades de sistemas clonados [Lima, et al. 2013]. Essa abordagem busca permitir que se continue a usar um processo tradicional de desenvolvimento, mas que possa-se integrar pelo menos parte de duas funcionalidades que são similares entre clones de um mesmo sistema. A Figura 1-2 exemplifica um cenário de reconciliação entre diferentes versões de clones de um mesmo sistema. Inicialmente o sistema original (Source System) foi clonado. Após esse processo ele continuou evoluindo o seu código fonte, e, ao mesmo tempo, o sistema clone (Target System) também evoluiu com novas funcionalidades. Em um determinado instante no futuro, deseja-se trazer para o sistema clonado, um conjunto de funcionalidades que foram desenvolvidas no sistema original, gerando então uma versão reconciliada do sistema *Target*.

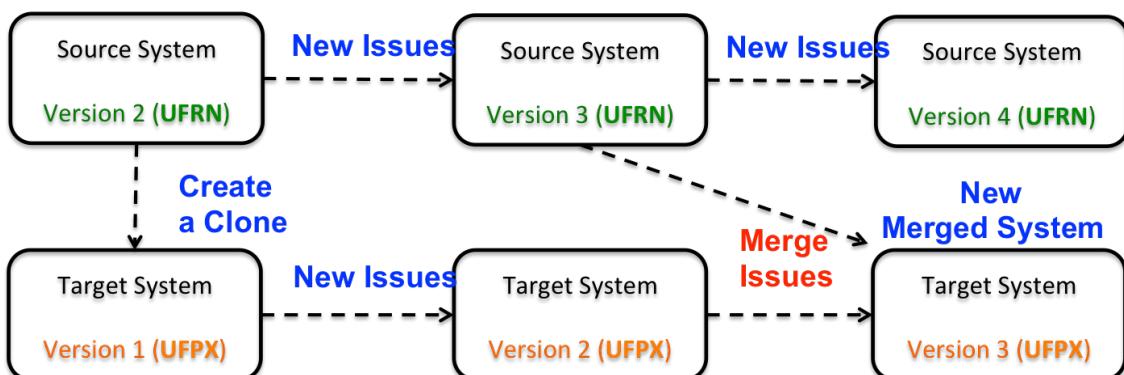


Figura 1-2: Reconciliação de funcionalidades entre clones

Embora estudos empíricos recentes [Guimarães & Silva. 2012], [Brun et al. 2013] já tenham dado uma perspectiva dos diferentes conflitos de *merge* que acontecem em sistemas de código aberto existentes, não há nenhum estudo empírico fornecendo uma visão detalhada quantitativa e qualitativa de como esses conflitos acontecem e a complexidade de integrá-los no contexto de sistemas comerciais clonados. Estas informações são importantes e podem ser usadas nos processos de desenvolvimento para organizar melhor o código e minimizar futuros esforços de integração.

1.3. Limitações dos Trabalhos Atuais

Vários trabalhos recentes abordam o problema de integração (*merge*) de funcionalidades de sistemas ou linhas de produto de software clonadas. Esta seção apresenta um breve panorama de tais trabalhos assim como destaca suas limitações. Guimarães & Silva [Guimarães & Silva. 2012] desenvolveram uma abordagem de *merge* contínuo que analisa os conflitos que vão sendo gerados a medida em que o desenvolvedor altera o código fonte do sistema no seu ambiente de trabalho (*workspace*) local. O objetivo dessa análise é antecipar a resolução de conflitos oriundos da alteração de partes similares do código e antes que o desenvolvedor submeta as suas mudanças para o repositório de código, onde os demais desenvolvedores também estão realizando alterações. A abordagem desenvolvida por tais autores se aplica a um contexto diferente do nosso, mas várias das análises e classificações de conflitos realizadas podem ser aplicadas ao nosso problema. A análise de conflitos da abordagem investigada nesta dissertação acontece a partir da construção e comparação de uma árvore de elementos da linguagem (pacotes, classes, atributos e métodos) mais simples do que a árvore de sintaxe abstrata (AST- *Abstract Sintax Tree*), do ambiente de trabalho (*workspace*) local do desenvolvedor com a versão remota (publicada no repositório) da árvore.

Rubin e outros [Rubin et al, 2012] propõem uma abordagem que busca explorar os benefícios da técnica de clonagem e mitigar as desvantagens do uso desse mecanismo. Para isso, eles definem um modelo chamado de *Product Line Changeset Dependency Model* (PL-CDM). O PL-CDM armazena grupos de modificações relacionadas do código fonte em funcionalidades e esses grupos de funcionalidades em grupos de *features*, além de relações de dependência entre essas funcionalidades e as *features* presentes no PL-CDM. A abordagem tem algumas semelhanças com a analisada nesse

trabalho [Lima, et al. 2013], principalmente no que diz respeito a geração de um modelo a partir de sistemas de gerência de configuração e mudanças. Porém, tal abordagem não foi ainda completamente implementada, apenas é sugerido maneiras de desenvolver o modelo proposto. Também não é mencionado nenhum tipo de análise ou classificação de conflitos, nem tampouco é apresentada alguma avaliação em um sistema em escala comercial.

Antkiewicz e outros [Antkiewicz, et al. 2014] apresentam uma estratégia minimamente invasiva para adoção da engenharia de linha de produtos de software. Eles propõem seis níveis de governança como um roteiro gradual de adoção eliminando transações custosas, perturbadoras e com altos riscos entre a abordagem *clone-and-own* e abordagem tradicional de engenharia de LPS. Os autores apresentam essa estratégia baseada na experiência de alguns casos da indústria, porém não apresentam nenhuma ferramenta que ajude a lidar com produtos clonados.

Lima [Lima, 2014] propõe uma abordagem para reconciliação de sistemas clonados a partir da mineração de informações das evoluções contidas em sistemas de gerenciamento de configurações e mudanças e análise de três tipos de conflitos para indicar possíveis problemas que podem ocorrer nessa reconciliação. Essa dissertação evolui a proposta apresentada em [Lima, et al, 2014], implementando algumas melhorias na versão da ferramenta que foi desenvolvida, além de propor um novo estudo utilizando essa ferramenta, para melhor avaliá-la.

1.4. Objetivos

Este trabalho apresenta um estudo exploratório que quantifica e analisa a complexidade de integrar conflitos de *merge* existentes em um sistema web de larga escala que usa a clonagem como forma de gerenciar variabilidades nas suas funcionalidades. Essa análise é realizada aplicando a abordagem proposta por [Lima, et al. 2013]. Os objetivos de nosso estudo são: (i) compreender os tipos de conflitos que acontecem quando evoluindo sistemas clonados e a dificuldade de integrá-los de um sistema clonado para outro; e (ii) avaliar qualitativamente a abordagem proposta por [Lima, et al. 2013].

1.5. Questões de Pesquisa

Para os objetivos propostos foram determinadas três questões de pesquisa para avaliar os conflitos gerados na realização da integração de funcionalidades clonadas além de avaliar a abordagem de reconciliação utilizada nesse trabalho:

Questão de Pesquisa 1: Tarefas identificadas pela abordagem como não tendo conflitos podem de fato ser integradas?

Pretende-se com essa questão de pesquisa avaliar se a análise dos tipos de conflitos propostos pela abordagem [Lima, et al. 2013] é suficiente para a integração de sistemas clonados.

Questão de Pesquisa 2: Tarefas indicadas com conflitos diretos [Lima, et al. 2013] realmente representam problemas de integração?

O objetivo dessa questão de pesquisa é avaliar a qualidade dos resultados da análise de conflitos diretos e verificar a complexidade de integração desse tipo de conflitos proposto.

Questão de Pesquisa 3: Os conflitos indiretos identificados pela abordagem [Lima, et al. 2013] representam conflitos que trazem problemas para a integração?

Procura-se com essa questão de pesquisa descobrir o quanto a estratégia de identificação de conflitos indiretos utilizando grafo de chamadas é eficiente com relação a identificação de problemas de integração de tarefas.

1.6. Organização do Documento

O restante dos capítulos desta dissertação estão organizados da seguinte maneira:

- i. O Capítulo 2 apresenta a fundamentação teórica relacionada ao tema desta dissertação e as tecnologias utilizadas para a sua implementação.
- ii. O Capítulo 3 descreve a abordagem utilizada no estudo e as melhorias realizadas para esta dissertação em relação a versão anterior.
- iii. O Capítulo 4 descreve um estudo empírico realizado para avaliar a abordagem implementada.

- v. O Capítulo 5 discute os trabalhos relacionados.
- vi. Por último, no Capítulo 6 apresenta as conclusões e trabalhos futuros.

2. Fundamentação Teórica

Neste capítulo é apresentada a fundamentação teórica desse trabalho. Na Seção 2.1 são discutidos os termos teóricos da área de Linhas de Produtos de Software. A Seção 2.2 apresenta um resumo sobre os sistemas de gerenciamento de configuração e mudança utilizados na fase mineração de evoluções de código da abordagem. A Seção 2.3 apresenta a parte teórica sobre mineração de repositórios também utilizada na fase de mineração das evoluções de código da abordagem. Por último, a Seção 2.4 discute a funcionalidade de *merge* de código fonte em sistemas de controle de versão.

2.1. Linhas de Produtos de Software

Quando se tem uma boa solução de software para atender a um segmento específico de mercado, muitos potenciais clientes se interessam por essa solução. Muitas empresas perceberam que seria uma grande vantagem competitiva se pudessem reusar suas soluções adaptando-as, de forma organizada, as especificidades de cada cliente.

Para promover esse reuso dos componentes de um software surgiu a abordagem de Linha de Produtos de Software (LPS). A engenharia de Linha de Produtos de Software foca não mais no desenvolvimento de um único sistema, mas em vários sistemas (produtos) que podem ser gerados a partir de uma base de componentes adaptáveis (linha). Essa mudança de foco implica também em uma mudança de estratégia de negócio da empresa, deixando de se preocupar apenas com o contrato com o próximo cliente e passando a ter uma visão estratégica em todo o campo do negócio que a linha de produtos pode alcançar.

Devido ao reuso em larga escala, Linhas de Produtos de Software melhoram a maioria das métricas no processo de desenvolvimento de software, reduzindo os custos, o tempo para o produto ser disponibilizado (*time to market*) e melhorando a qualidade dos produtos resultante, como por exemplo a sua confiabilidade. Segundo [Van der Linden, F. J. Schmid, K. Rommes, E. 2007], essa redução pode ser de até 90% nos custos e tempo de desenvolvimento se comparado aos processos tradicionais de desenvolvimento do software.

Contudo, a aplicação da abordagem de LPS também gera algumas desvantagens e problemas que precisam ser gerenciados. Como mostrado na Figura 2-1, LPS requer um

investimento inicial maior, normalmente a estratégia de LPS passar a ser vantajosa financeiramente a partir do desenvolvimento de terceiro produto. Isso pode ser um problema para empresas pequenas que não tenham aporte financeiro suficiente.

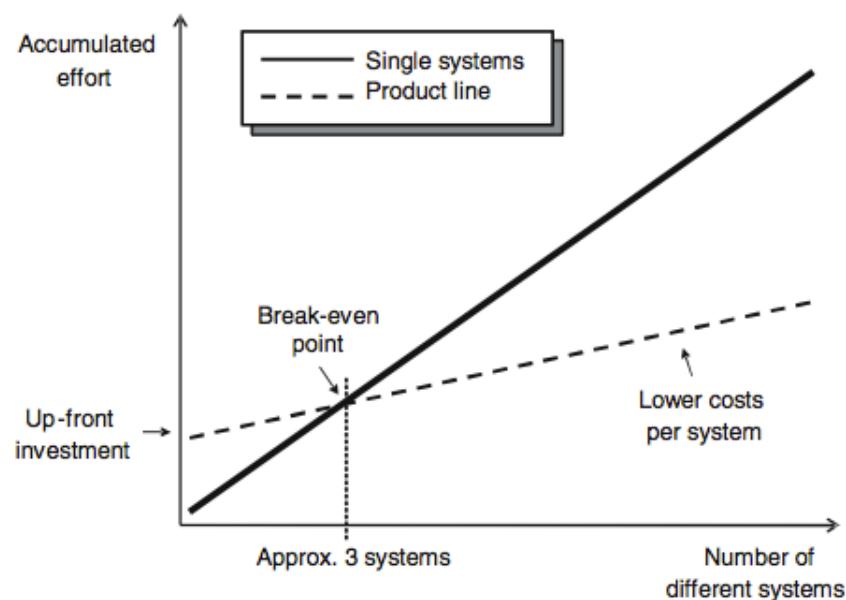


Figura 2-1: Economia no uso de Linha de Produtos de Software
Fonte: [Van der Linden, F. J. Schmid, K. Rommes, E. 2007]

Além disso, LPS requerem uma engenharia experiente e um maior gerenciamento técnico e organizacional, pois a complexidade do desenvolvimento aumenta, haja visto que não mais um único sistema está sendo desenvolvido, mas vários produtos para atenderem vários clientes, cada um com um conjunto variações nas suas regras de negócio.

Por último, LPS requerem um domínio estável de negócio. Como a principal finalidade de uma LPS é promover o reuso em massa voltado ao negócio da organização, fica quase impossível promover esse reuso se o domínio de negócio está mudando o tempo todo.

2.1.1 Similaridades e Variabilidades

Os conceitos de similaridades e variabilidade são os conceitos chaves em linhas de produtos.

- Similaridade: Uma característica (funcional ou não funcional) que é comum a todos os produtos da linha.

- Variabilidade: Uma característica que pode variar de um produto para outro. Uma variabilidade deve ser modelada de forma que possa ser selecionada para estar presente em apenas alguns produtos.

2.1.2 Features

Uma *feature* pode ser definida como uma característica essencial das aplicações de um domínio [Kang et al., 1998]. Representam uma característica do sistema visível pelo usuário final, ou seja estão relacionadas às regras de negócio da aplicação. Uma *feature* pode ocorrer em qualquer nível, desde dos requisitos, passando pela arquitetura até a parte de implementação do sistema.

Uma *feature* pode ser usada para representar partes comuns e variáveis do sistema. Normalmente a engenharia de Linhas de Produtos é focada nas *features* variáveis, que são as responsáveis por distinguirem um produto de outro da linha. Para documentar as *features* existentes em uma Linha, usualmente utiliza-se o modelo de *features* ou *Feature Model* (FM). *Feature Model* é um modelo em forma de árvore utilizado para modelar variabilidades de uma linha de produtos. Muitas representações foram propostas, uma das mais utilizadas é a representação Feature-Oriented Domain Analysis (FODA). A Figura 2-2 mostra os tipos de *features* utilizando a representação FODA.

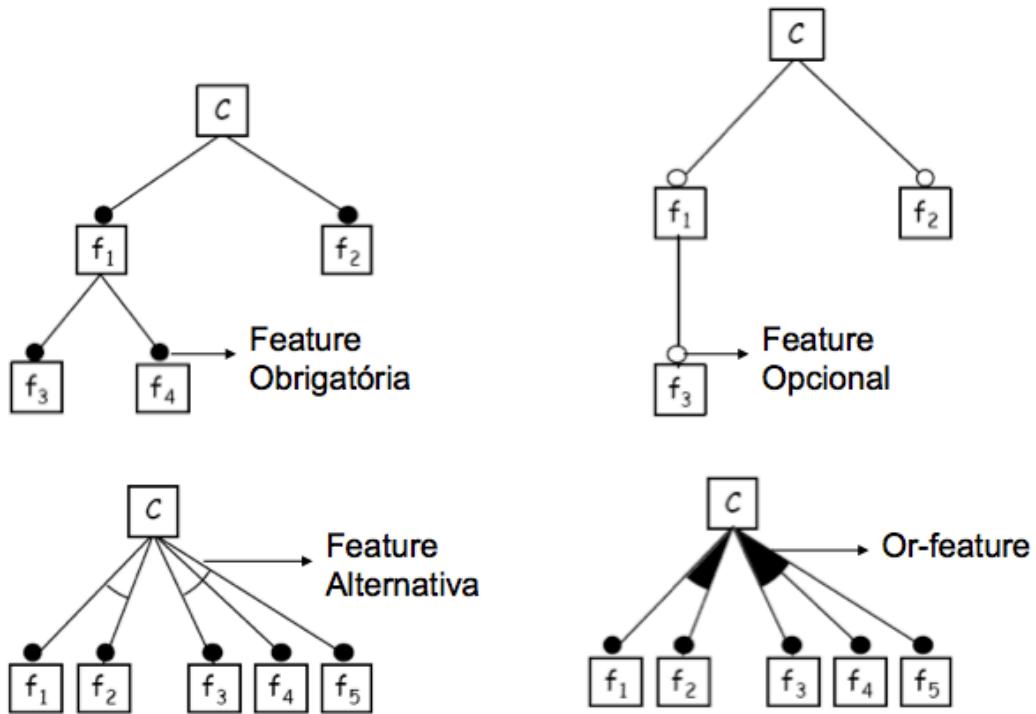


Figura 2-2: Representações dos tipos de Features usando a representação FODA
Fonte: [NUNES, 2010]

Existem quatro tipos principais de *Features*:

- Obrigatórias: Estão em todos os produtos.
- Opcionais: Podem ou não estarem em um produto.
- Alternativas: exatamente uma das *features* deve estar no produto.
- *Or-features*: um subconjunto das *features* pode estar no produto.

2.1.3 Processo de Derivação do Produto

O processo de seleção de quais *Features* estarão presentes em um determinado produto é denominado Derivação do Produto. Para esse processo ser realizado, normalmente é mantido um mapa entre as *features* e os artefatos correspondentes a essas *features*. Esse mapeamento é denominado de *Configuration Knowledge* (CK). Com as informações contidas no CK é possível determinar quais artefatos devem estar presentes em um produto da linha a partir das *features* selecionadas.

2.1.4 Técnicas de Implementação de Variabilidades

Existem 3 técnicas básicas para implementar variabilidades em linhas de produtos, conforme mostrado na Figura 2-3:

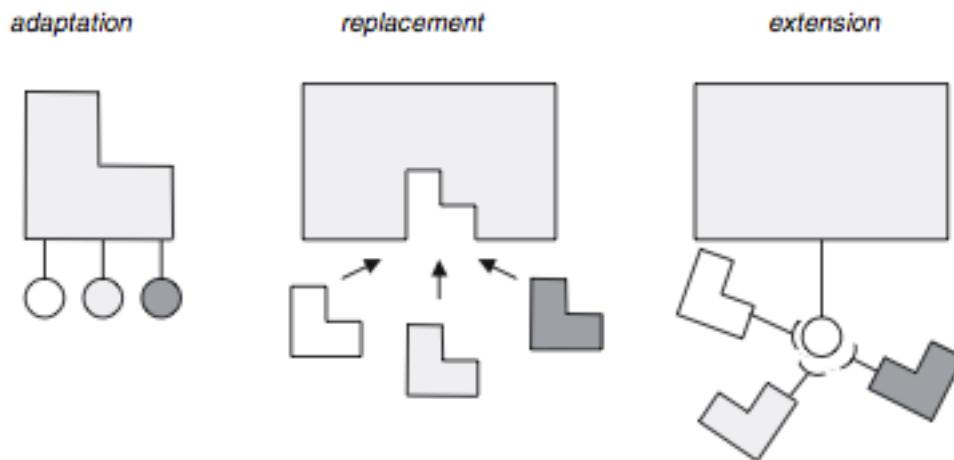


Figura 2-3: As técnicas básicas de realizar variabilidades em Linhas de Produtos de Software
Fonte: [Van der Linden, F. J. Schmid, K. Rommes, E. 2007]

- Adaptação: Na adaptação é fornecido apenas uma única implementação para um certo componente, porém esse componente fornece interfaces para que o seu comportamento possa ser alterando.
- Substituição: Várias implementações de um componente estão disponíveis. Uma dessas implementações é escolhida para fazer parte do produto gerado.
- Extensão: A arquitetura da linha de produtos oferece uma interface genérica que permitem adicionar novos componentes ao produto. Permitindo vários tipos de componentes diferentes.

Existem vários mecanismos concretos que podem ser usados para implementar variabilidades em uma linha de produtos. Como alguns dos principais mecanismos pode-se destacar:

- **Herança:** O comportamento de uma classe pode ser estendido para se adicionar novos comportamentos ou sobreescriver outros.
- **Configuração em tempo de compilação:** compiladores podem oferecer mecanismos de variações do componente no tempo de compilação. Trechos de código são marcados com certas condições, um pré-

processador analisa o código e retira os trechos que não devem ser incluídos de acordo com os valores das condições. Essa técnica também é conhecida como Compilação Condicional.

- **Configuração em tempo de execução:** Trechos de código são executados ou não dependendo de certas condições que são avaliadas em tempo de execução. Essa técnica também é conhecida como Execução Condicional [Santos, J., et al. 2012].
- **Aspectos:** Trechos de código são introduzidos, de acordo com determinadas condições, em lugares previamente configurados em tempo de compilação ou execução por meio de Aspectos [Soares, A., et al. 2015].
- **Geração de Código:** a geração de código lê algum tipo de especificação de alto nível e gera o código requerido para implementar uma determinada função.
- **Plug-ins:** uma arquitetura oferece interfaces que permitem plugar componentes ao sistema. Os componentes conectados fornecem novas funcionalidades estendendo os comportamentos padrões.

2.1.5 Fases da Engenharia de Linhas de Produtos

A engenharia de Linhas de Produtos de Software é comumente dividida em duas grandes fases: Engenharia de Domínio e Engenharia de Aplicação, conforme mostrado da Figura 2-4.

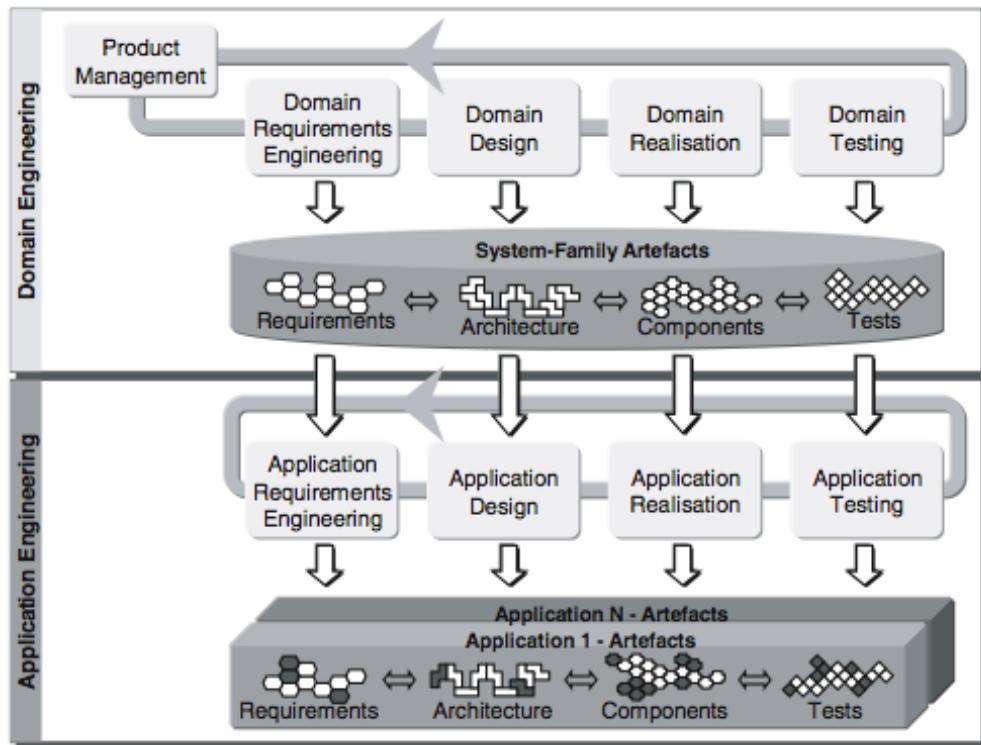


Figura 2-4: Ciclos de vida da Engenharia de Linhas de Produtos de Software

Fonte: [Van der Linden, F. J. Schmid, K. Rommes, E. 2007]

A Engenharia de Domínio é voltada para a implementação dos artefatos de software. Esse artefatos cobrem todas as fases de tradicionais de desenvolvimento e um sistema, desde os requisitos, passando pela elaboração da arquitetura e implementação até a fase de testes. Todos os artefatos implementados nessa fase formam a infraestrutura da linha de produtos, que na próxima fase permitirá a derivação dos produtos da linha, por isso essa fase é normalmente mencionada como desenvolvimento para reuso.

O foco na Engenharia de Aplicação é o desenvolvimento do produto final utilizando a infraestrutura da linha de produtos. As variabilidades modeladas na infraestrutura permitem a criação de um produto específico, por isso essa fase é normalmente mencionada como desenvolvimento com reuso.

Existem também três principais abordagens de uma LPS:

- **Proativa:** Projetar o sistema desde o começo para ser uma linha de produtos, com todas as suas variabilidades. Requer grande investimento inicial. Esse tipo de abordagem é usada normalmente por grandes empresas.

- **Extrativa:** As *features* já existem no produto já utilizado, então elas são extraídas para se criar a linha.
- **Reativa:** A medida que surge a necessidade, as *features* são criadas gerando-se um novo produto na linha.

2.2. Gerenciamento de Configuração e Mudanças

Gerenciamento de Configuração e Mudanças (GCM) é um conjunto de práticas realizadas com o objetivo de controlar e auditar o processo do desenvolvimento e manutenção de um produto. [Berczuk, S., Appleton, B. 2002.] cita que: práticas de GCM tomadas em conjunto definem como uma organização constrói e disponibiliza seus produtos, e identifica e controla as alterações.

A principal ferramenta do GCM é Controle de Versões. Ele controla as diferentes versões dos artefatos durante o processo de desenvolvimento do software, tais como: códigos-fonte e documentos. Além disso, mantém um histórico completo de todas as alterações efetuadas, possibilitando ainda a comparação entre versões, identificação dos responsáveis pelas alterações, marcação de versões específicas e ramificações do projeto.

Entre os principais sistemas de controle de versão (SCV) mais utilizados atualmente para realizar o controle de versão estão o Subversion e o GIT.

- Subversion ou SVN é um sistema de controle de versão *open-source*¹, cliente-servidor, que tem por objetivos gerenciar arquivos e diretórios, e as modificações feitas neles ao longo do tempo.
- GIT é um sistema de controle de versão distribuído *free*² e *open-source* projetado para lidar com pequenos ou grandes projetos com velocidade e eficiência.

Os SCV estão sujeitos a conflitos de edição. Essa situação ocorre no momento em que dois ou mais usuários modificam o mesmo documento no mesmo intervalo de tempo. Apesar da eficiência em termos de desempenho, uma das deficiências na

¹ Termo usado para se referir a sistemas que disponibilizam o seu código fonte.

² Sistemas disponibilizados sem custos ou licenças de uso.

³ Sistema que divide o código fonte em partes menores, dividindo-o em diretórios e subdiretórios.

utilização dos SCV é que esses sistemas são orientados a texto, isto é, quando um conflito ocorre, eles apenas destacam as linhas do arquivo que foram alteradas em paralelo com as linhas da versão que está no repositório. É de responsabilidade do desenvolvedor resolver o conflito. Não é realizada nenhuma análise do impacto que a resolução do conflito possa ter causado no sistema. Normalmente apenas conflitos simples podem ser resolvidos utilizando o suporte fornecido pelos SCV, em situações mais complexas, os SCV não oferecerem o auxílio necessário, o que torna essa tarefa de resoluções de conflitos bastante árdua.

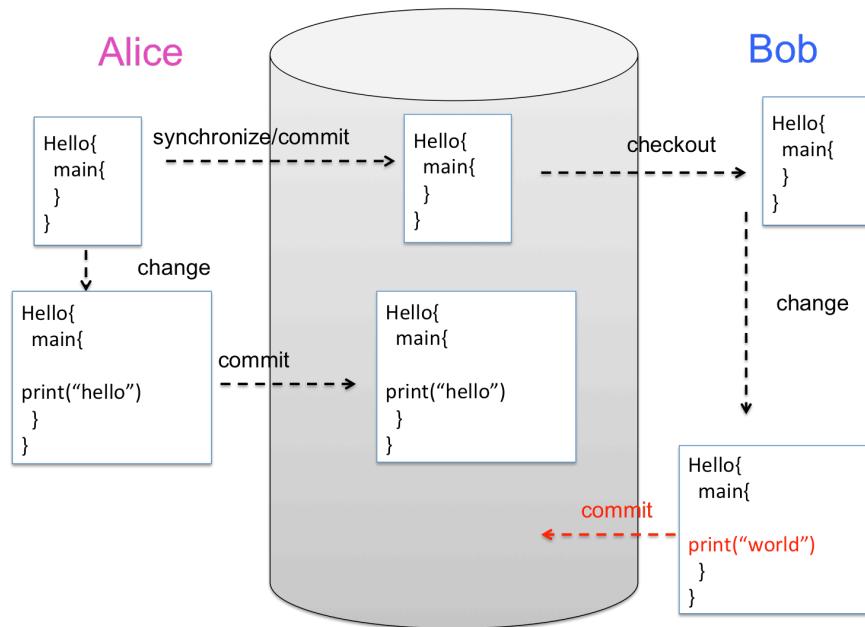


Figura 2-5: Exemplo de Conflito de Edição

A Figura 2-5 mostra um exemplo de conflito que ocorre quando os usuários Alice e Bob realizam mudanças em um mesmo artefato de código fonte de maneira paralela. Alice envia primeiro a sua mudança para o sistema de controle de versão, quando Bob tenta, em um momento futuro, submeter a mudança dele ao controle de versão, essa mudança entra em conflito com a mudança realizada previamente por Alice. Neste caso, Bob deve resolver o conflito gerado antes de poder submeter a sua mudança ao sistema de controle de versão.

Existe algumas boas práticas como a integração contínua [Fowler, M. 2006] que visa, entre outras coisas, diminuir a quantidade de conflitos gerados, porém em

determinados cenários, como os cenários que levam a geração de clones de um sistema, os conflitos não podem ser evitados.

2.3. Mineração de Repositórios

Os repositórios são depósitos de informação sobre as atividades de desenvolvimento e de manutenção de um software. Essas informações vão desde o código fonte, sistema de configuração e mudança ou ferramentas de comunicação. Como exemplos de repositórios pode-se citar os sistemas de controle de versão (CVS, Subversion, Git, Mercurial, etc.), sistemas de configuração e mudança (Bugzilla, ClearQuest, Redmine etc.) e bancos de dados (PostgreSQL, Oracle, MySQL, etc.).

A área de Mineração de Repositório (MRS) se preocupa em extrair dados disponíveis de repositórios de softwares para descobrir informações relevantes sobre sistemas de informações e projetos de software [HASSAN, 2008] citado por [Alencar, D. 2013].

A mineração de repositórios consiste em 3 etapas:

- Identificação de quais repositórios podem ser úteis e que tipo de informações eles guardam.
- Realizar minerações nos repositórios de software com o objetivo de investigar um determinado problema.
- A análise dos resultados obtidos com a mineração.

A ferramenta avaliada nesse trabalho utiliza mineração de repositório para extrair informações sobre evoluções sofridas em paralelo nos clones gerados para um sistema.

2.4. Merge de Código Fonte

Merge ou integração é uma operação fundamental que concilia múltiplas alterações feitas em uma coleção de arquivos que estejam “*versionados*” em um sistema de controle de versão. Na maioria das vezes, essa operação é necessária quando um mesmo arquivo de código fonte é modificado por duas pessoas distintas concorrentemente. O resultado da operação de *merge* é uma única coleção de arquivos que contém um subconjunto das alterações realizadas em ambos os lados.

Em alguns casos essa operação pode ser realizada automaticamente, porque há informações históricas suficientes para reconstruir as mudanças realizadas e elas não entram em conflito, neste caso é dito que ocorreu um Merge Automático. Em outros casos, o mesmo trecho de código fonte foi alterado e as mudanças possuem conflitos, não podendo coexistirem. É necessária uma intervenção humana para decidir qual das mudanças os arquivos resultantes da operação de *merge* devem conter.

2.4.1 Two-Way Merge

É um algoritmo de *merge* que verifica apenas duas cópias do arquivo modificado. Com esse tipo de algoritmo não é possível identificar a alteração que foi realizada, pois não se tem nenhuma informação de como o arquivo se encontrava antes da modificação. A Figura 2-6 mostra um exemplo das informações analizadas por um algoritmo de Two-Way Merge.

Yours	Mine
28	
29	
30 print("hello");	30 print("world");
31	
32	

Figura 2-6: Two-Way Merge
Fonte: [SANTOS, 2013]

2.4.2 Three-Way Merge

O algoritmo Three-Way Merge procura compensar a deficiência do algoritmo Two-Way Merge verificando como o arquivo estava originalmente antes da realização das mudanças. A Figura 2-7 apresenta uma esquematização do algoritmo Three-Way Merge. Como pode ser percebido, com base em um ancestral comum é possível identificar que a alteração ocorreu na linha 30 destacada em verde. Esse algoritmo possibilita que alguns problemas de eram de resolução manual, utilizando o algoritmo Two-Way Merge, sejam tratados como um *merge* automático.

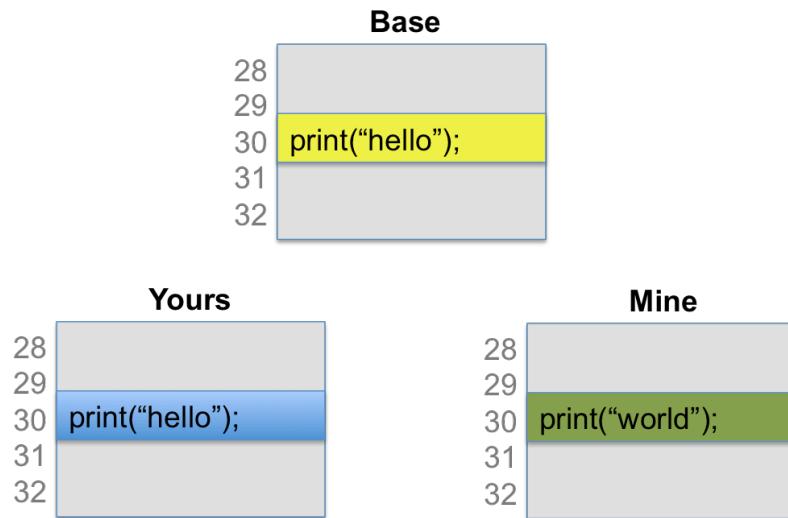


Figura 2-7: Three-Way Merge

Fonte: [SANTOS, 2013]

3. Abordagem de Merge para Sistemas Clonados

Este capítulo apresenta a abordagem proposta por [Lima, et al. 2013] para realizar a análise de conflitos em sistemas web clonados. A abordagem é utilizada neste trabalho para realizar o estudo empírico proposto. O presente capítulo também destaca as melhorias realizadas com relação à versão anteriormente implementada da abordagem. A Seção 3.1 apresenta uma visão geral da ferramenta. A Seção 3.2 descreve o módulo de mineração das evoluções ocorridas. A Seção 3.3 descreve como ocorre a identificação dos conflitos. A Seção 3.4 mostra a estratégia de resolução de conflitos sugerida pela abordagem. Finalmente a Seção 3.5 apresenta as evoluções técnicas realizadas na abordagem, parte da contribuição desta dissertação.

3.1. Visão Geral

A abordagem para *merge* de sistemas clonados proposta por [Lima, et al. 2013] consiste de quatro etapas principais: (i) minerar as evoluções de sistemas web clonados e organizá-las de forma estruturada; (ii) identificar diferentes tipos de conflitos ao realizar o *merge* de tarefas entre dois sistemas web clonados; (iii) a partir da identificação desses conflitos, indicar qual estratégia de *merge* deve ser utilizada; e (iv) aplicar a estratégia de *merge* indicada e executar testes para garantir que a reconciliação não adicionou erros ao sistema. Até o presente momento, o objetivo (iv) da abordagem não chegou a ser implementado e não será avaliado nesse trabalho. Para essa abordagem foi desenvolvida uma ferramenta, denominada MergeClear [Santos, J., et al. 2015], que implementa a abordagem proposta por [Lima, et al. 2013]. A Figura 3-1 mostra uma visão geral do processo de *merge* dessa ferramenta.

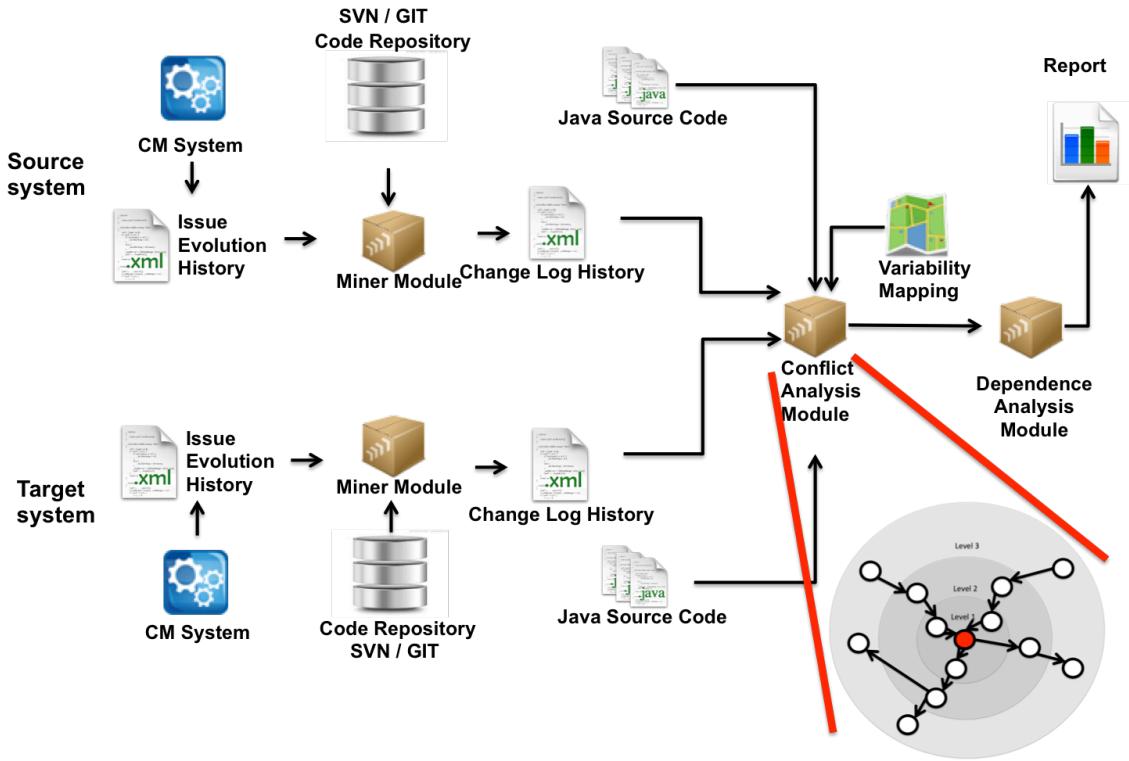


Figura 3-1: Visão Geral do Processo de Merge entre sistemas web clonados

A abordagem foi concebida para analisar sempre um par de clones por vez. O *Source system* é o sistema que contém as evoluções que se deseja aplicar, o *Target system* é o sistema no qual se deseja aplicar as evoluções ocorridas em paralelo. A ferramenta inicia o processo lendo informações das evoluções de um sistema a partir de *logs* registrados nas tarefas (*tasks*) do sistema de gerência de mudanças (*issue tracker*). Em seguida, ela acessa o repositório do sistema de controle de versão para extrair informações detalhadas sobre a evolução dos artefatos de código.

A partir daí, ela organiza essas informações em dois modelos estruturados que contém o histórico de evoluções ocorridas no sistema *Source* e no sistema *Target*. Essas informações juntamente com o código Java e um mapeamento de variabilidade são a entrada para a fase de análise de conflitos. Dependendo dos tipos de conflitos encontrados, uma estratégia diferente de *merge* pode ser utilizada para reconciliar funcionalidades dos sistemas.

O mapeamento de variabilidades é um conhecimento de configuração [Czarneck, Eisenecker. 2000] que, quando fornecido, armazena o mapeamento de variabilidades para artefatos de código fonte da LPS. Ele também é utilizado para

organizar as informações da evoluções identificadas pela ferramenta em termos de *features*. A abordagem foi inicialmente desenvolvida para reconciliação de LPSs clonadas, contudo nada impede que a ferramenta desenvolvida seja aplicada em outros contextos como na realização do *merge* entre duas *branches*³ de um mesmo sistema que não seja uma LPS. Neste caso, tem-se a possibilidade de não fornecer o mapeamento de variabilidades, para que as evoluções não sejam associadas a *features*

A ferramenta foi desenvolvida usando a plataforma do Eclipse com o objetivo de permitir a mineração de sistemas web implementados na linguagem em Java. A arquitetura da ferramenta foi projetada de forma flexível, podendo ser estendida para suportar novas tecnologias, como, por exemplo, a substituição do mecanismo padrão para geração do grafo de chamada, sem que isso afete as classes que utilizam esse grafo. Outro ponto que a flexibilização da arquitetura da ferramenta ajuda é na integração da ferramenta com outros sistemas de gerenciamento de configuração e mudança, e outros sistemas de controle de versão que não os suportados por default.

Os dois principais módulos da ferramenta são o módulo de mineração (*Evolution Mining Module*) e o módulo de análise de conflitos (*Conflict Analysis Module*). A *Figura 3-2* apresenta uma visão geral da arquitetura da ferramenta.

³ É uma ramificação no desenvolvimento, usada para descrever o processo de divisão dos arquivos de um projeto em linhas de desenvolvimento independentes. Podendo servir para teste de uma nova funcionalidade ou para projetos destinados a um cliente específico.

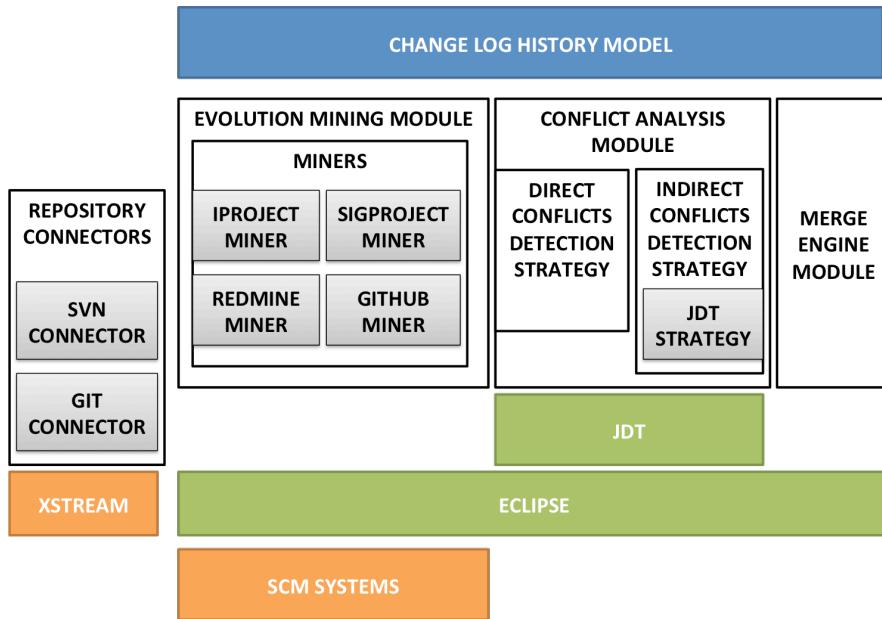


Figura 3-2: Visão Geral da Arquitetura do MergeClear

O *Evolution Mining Module* é composto de vários *Miners*. Cada Miner foi desenvolvido para realizar a mineração das informações do sistema de gerenciamento de configuração e mudança específico, visto que não existem um padrão de como as informações sobre a evolução de um sistema devem ser documentadas nesses sistemas.

O *Conflict Analysis Module* é dividido em dois sub módulos principais: (i) o *Direct Conflicts Detection Strategy* que analisa os conflitos diretos gerados na evolução; e (ii) o *Indirect Conflict Detection Strategy*, que contém o algoritmo de análise dos conflitos indiretos gerados nas evoluções analisadas.

Caso um desses módulos, ao precisarem acessar informações sobre a evolução do código fonte, utilizam o módulo *Repository Connectors*, que possui sub módulos para oferecer suporte para os sistemas de controle de versões específicos onde o código dos sistemas analisados estejam armazenados.

O resultado da mineração junto com a análise de conflitos são armazenados em um modelo estruturado chamado *ChangeLog History Model*. A partir desse ponto, as demais etapas devem utilizar as informações contidas nesse modelo, tornando assim, essas próximas etapas independente dos softwares utilizados no processo de configuração e mudanças dos sistemas analisados.

3.2. Módulo de Mineração das Evoluções

A primeira etapa da abordagem é a mineração das informações relacionadas à evolução dos dois sistemas clonados. A intenção da mineração é compilar o máximo de informação possível para ajudar o desenvolvedor a realizar o *merge* entre as duas versões de sistemas clonados, não se restringindo apenas à informações sobre as evoluções ocorridas no código fonte. Para isso, o *MergeClear* inicia o processo de mineração extraíndo informações sobre as tarefas desenvolvidas entre as versões (inicial e final) que se deseja analisar dos dois sistemas evoluídos em paralelo (*Source* e *Target*). As informações dessas tarefas são extraídas dos sistemas de gerência de configuração e de mudanças (*SCM Systems*) que são utilizados pela equipe de desenvolvimento do sistema. Exemplos de informações coletadas pela ferramenta vão desde a descrição da tarefa até a sua prioridade ou em qual release ela foi disponibilizada.

A integração com esses sistemas é realizada a por meio de classes denominadas Miners e Connectors. A Figura 3-3 apresenta um diagrama de classes da arquitetura do *MergeClear*.

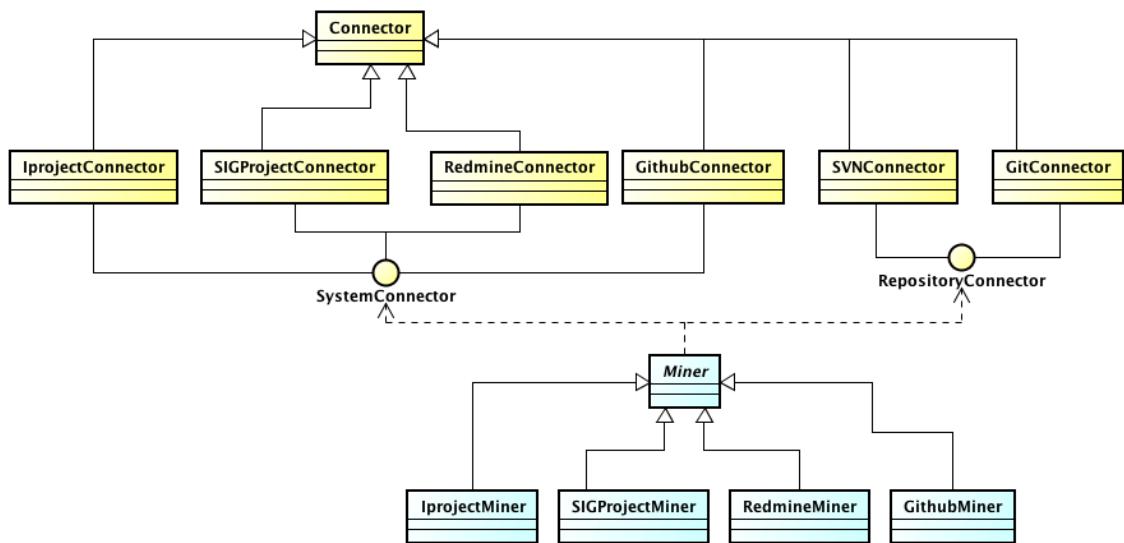


Figura 3-3: Diagrama de Classes do *MergeClear*

Um Connector representa uma classe de conexão com sistemas externos. Um Connector pode ser de dois tipos: (i) SystemConnector que representa uma conexão

com um sistema de gerenciamento de configuração e de mudanças e (ii) RepositoryConnector que representa uma conexão com repositório de código fonte.

Um Miner representa a classe que executa a mineração das evoluções. Ela possui sempre relação com dois Connectors: (i) um SystemConnector para extrair informações sobre a evolução oriundas do processo de desenvolvimento, normalmente essas informações são informações das tarefas realizadas por um desenvolvedor, com suas respectivas descrições, prioridades, tipos, entre outras informações e (ii) um RepositoryConnector para recuperar informações sobre a evolução do código fonte em si.

Atualmente a ferramenta oferece suporte para integração com quatro sistemas de gerência de configuração e mudanças (Iproject, SigProject, RedMine e GitHub) e dois repositórios de código fonte (SVN e GIT). Suportando assim, 8 configurações diferentes. É possível, por exemplo, minerar informações de evolução de um sistema *Source* que possua seu código no SVN e esteja utilizado o RedMine como sistema de gerência de configuração e mudanças, com um sistema *Target*, que tenha seu código versionado no GIT e cujas informações do desenvolvimento estejam armazenadas no Iproject.

Caso seja necessário realizar a integração com um novo sistema de gerência de configuração e de mudanças ou um novo repositório de código fonte, apenas um novo SystemConnector e um novo RepositoryConnector precisam ser escritos. Em caso de ser necessário um novo algoritmo de mineração para tratar informações diferentes dos sistemas já suportados, um novo Miner será necessário. O suporte aos sistemas de configuração e mudanças RedMine e GitHub, além do repositório GIT foram adicionados na versão da ferramenta utilizada nesse trabalho, não estando presentes na versão anteriormente avaliada.

O próximo passo da mineração consiste em, para cada tarefa retornada, pelos sistemas de gerenciamento de configuração e de mudanças, recuperar os arquivos de código fonte que foram modificados nas respectivas tarefas. Recuperando os arquivos de código fonte, existem duas alternativas que podem ser utilizadas para detectar as artefatos alterados e organizar essa informação de forma estruturada levando-se em consideração a estrutura da linguagem para a qual o sistema que está sendo analisado foi

desenvolvido. A primeira é utilizar o algoritmo padrão da ferramenta que é descrito na Listagem 3-1.

Listagem 3-1. Algoritmo de detecção de mudanças

Compara o código fonte da classe da versão modificada com a versão anterior

Para cada campo da versão modificada

Verifica se na versão anterior existe um campo com o mesmo nome

Se existe e alguma outra informação, desconsiderando o nome, está diferente

Campo modificado

Se não existe

Campo adicionado

Para cada campo da versão anterior

Verifica se existe na versão modificada algum campo com mesmo nome

Se não existe

Campo apagado

Para cada método da versão modificada

Verifica se na versão anterior existe um método com o mesmo nome

Se existir método com mesmo nome

Verifica se alguma informação foi alterada (corpo do método, anotações, lançamento de exceções, parâmetros, modificadores)

Se sim

método foi modificado

Se não

Verifica se tem algum método com mesmo corpo, mas com nome diferente

Se existe método mesmo corpo

Método modificado

Se não

Método adicionado

Para cada método da versão anterior

Verifica se existe na versão modificada algum método com mesmo nome

Se não existe

Verifica se existe com mesmo corpo

Se não existe

Método apagado

A segunda alternativa é passar o código fonte modificado e a sua versão anterior para o *ChangeDistiller* [Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. 2007] que realiza essa análise e retorna as modificações entre a versão que evoluiu

dentro das versões analisadas do sistema e a sua versão anterior. No caso do uso do *ChangeDistiller*, a ferramenta se encarrega apenas de interpretar as informações retornadas e converter essas informações para o modelo padrão do *MergeClear*, descrito a seguir. Tal integração com o *ChangeDistiller* foi também realizada como parte dos trabalhos dessa dissertação. Após a utilização do *ChangeDistiller*, percebeu-se que o seu algoritmo apresentou resultados bastante semelhantes ao algoritmo implementado, com isso ele acabou não sendo utilizado para realizar os estudos desse trabalho, contudo ele ainda faz parte da ferramenta e pode ser utilizado se desejado.

Após esse processo de mineração, todas essas informações são armazenadas no modelo criado pela abordagem para guardar o histórico de evoluções ocorridas entre a versão inicial e final analisada do sistema *Source* e do sistema *Target*. Esse modelo é chamado *ChangeLogHistory* e sua estrutura é apresentada na Figura 3-4.

ChangeLogHistory Model

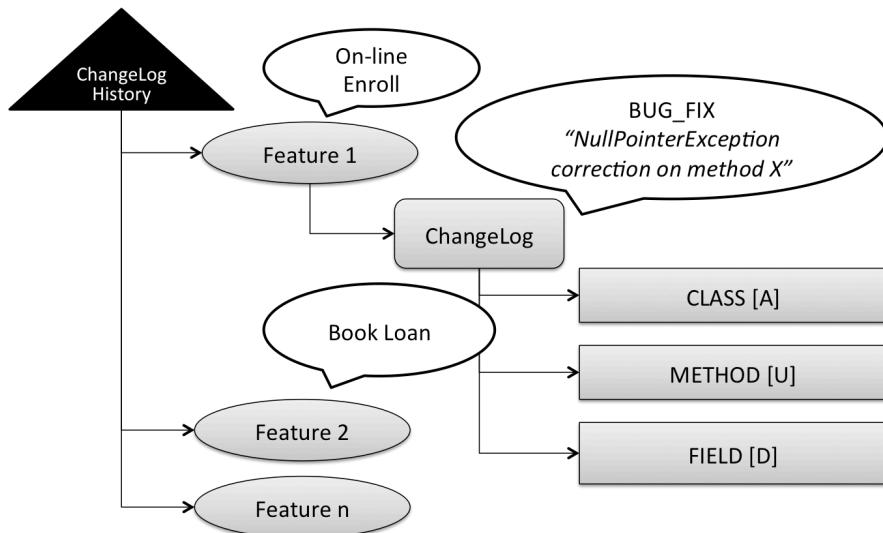


Figura 3-4: Modelo *ChangeLogHistory*

O modelo *ChangeLogHistory* organiza as informações em termos de *feature* da LPS, caso o mapeamento de *features* seja fornecido. Dentro de cada *feature* estão listados os elementos *ChangeLog* que representam uma mudança relacionada à *feature* pai. Normalmente um *ChangeLog* representa uma tarefa realizada pelo desenvolvedor que alterou o código fonte do sistema, sendo assim, ele contém a diferença na evolução do código fonte de cada artefato (classes, métodos, campos, entre outras informações da

linguagem Java) realizadas em todos os *commits* associados na tarefa. Essa diferença é organizada no modelo de forma estruturada, ou seja, não orientado à linha de código, mas sim ao tipo de alteração realizada no artefato. Como exemplo desse tipo de alteração pode-se citar: Criação de uma Classe, Alteração de um Método, Remoção de um campo, entre outras.

Esse modelo é salvo em arquivos XML utilizando a biblioteca XStream. A partir desse ponto, a análise de conflitos e a aplicação do *merge* foram pensadas para utilizar esse modelo e ficarem o máximo possível independentes do sistema de configuração e mudança ou da ferramenta de controle de versão da onde essas informação foram mineradas.

3.3. Módulo de Análise de Conflitos

Após finalizada a etapa de mineração e ter se gerado o modelo *ChangeLogHistory* para os sistemas *Source* e *Target*, a próxima etapa da abordagem é tentar identificar e classificar os conflitos que podem ocorrer ao tentar realizar o *merge* entre os dois sistemas evoluídos em paralelo.

Um conflito, para a abordagem, é definido como sendo um par de alterações no código fonte evoluídas em paralelo, que interferem entre si ao realizar o *merge* dessas alterações no código fonte do sistema *Source* para o sistema *Target*. Os tipos de conflitos analisados são descritos nas próximas subseções.

3.3.1 *Conflitos Diretos*

Conflitos que ocorrem quando o mesmo método ou atributo é alterado nos dois sistemas que estão sendo analisados, são classificados como conflitos diretos. Em outras palavras, o mesmo método ou atributo está presente no *ChangeLogHistory* do sistema *Source* e no *ChangeLogHistory* do sistema *Target*. A Figura 3-5 mostra um exemplo de um conflito direto.

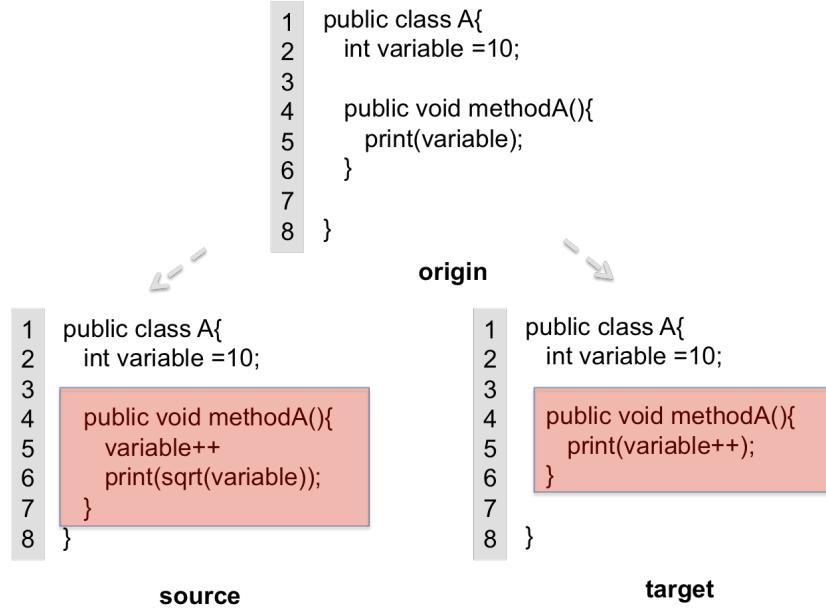


Figura 3-5: Exemplo de um Conflito Direto

A método `methodA` foi alterado nos dois lados da análise gerando um conflito direto. A Listagem 3-2 apresenta o algoritmo de detecção de conflitos diretos utilizado na abordagem.

Listagem 3-2. Algoritmo de detecção de conflitos diretos

Para cada artefato presente no ChangeLogHistory do sistema *Source*, verifique se ele está presente no ChangeLogHistory do sistema *Target*

Se está presente

Esses artefatos representam um conflito direto

3.3.2 Conflitos Indiretos

Uma alteração em um artefato que afete indiretamente outro artefato que está relacionado com o artefato alterado são classificados pela abordagem como conflitos indiretos. A Figura 3-6 mostra a representação de um conflito indireto.

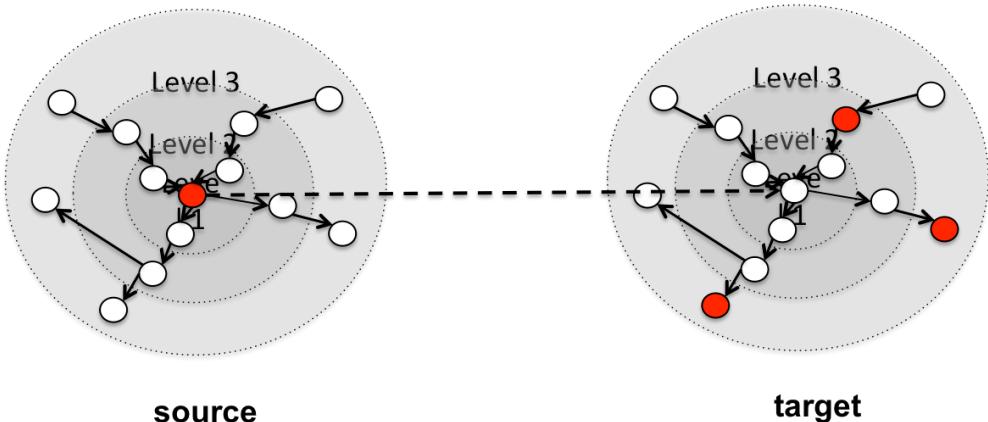


Figura 3-6: Representação de um Conflito Indireto

Nessa figura, artefatos (métodos ou atributos) marcados em vermelho representam uma evolução. Deseja-se aplicar as mudanças do artefato central do lado *Source* no artefato central do lado *Target*. O artefato que será integrado não apresenta mudanças no lado *Target*, porém artefatos relacionados com ele no grafo de chamadas sofreram alterações. A Listagem 3-3 apresenta o algoritmo de detecção de conflitos indiretos usado pela ferramenta.

Listagem 3-3. Algoritmo de detecção de conflitos indiretos

Para cada método ou campo presente no ChangeLogHistory do sistema *Source*, que **não** está em conflito direto

Calcula o grafo de chamadas desse método ou campo no sistema *Target*
 (Os métodos ou campos que fazem referência a ele ou os métodos ou campos que ele faz referência) recursivamente até o nível de profundidade configurado

Para cada dependência retornada verifica se essa dependência evoluiu, ou seja, se está presente do ChangeLogHistory do *Target*.

Se o método ou campo em si não sofreu alteração, mas algum método ou campo por ele referenciado sofreu alteração no *Target*

 Esse método ou campo está em conflito indireto

A abordagem considera que a análise de conflitos indiretos é executada após a análise de conflitos diretos. Os artefatos em conflitos diretos não são verificados nessa análise, pois para a abordagem, se eles apresentam conflitos diretos, já apresentam um

problema mais complexo de integração, o qual a abordagem não se propõem a resolver, por isso uma análise de conflitos indiretos é desnecessária.

Para identificar as referências de um artefatos e se gerar o grafo de chamadas da análise de conflitos indiretos utiliza-se o *Java Search Engine* [Wang, X. 2014] do JDT. Durante o desenvolvimento da ferramenta, foram verificadas outras ferramentas de análise estática como o WALA [IBM, 2015], porém além da dificuldade maior de configuração e uso de tal ferramenta, o WALA na sua configuração padrão, não suportou gerar o grafo de chamado para o sistema utilizado no estudo. O JDT permitiu um controle maior da análise e a possibilidade de implementar otimizações que suportassem a execução da análise de conflitos indiretos em um tempo aceitável.

3.3.3 Pseudo Conflitos

Esse conflitos são conflitos que ocorrem quando uma classe sofre alterações em ambos os sistemas clonados, mas os campos e métodos da classe que mudaram são distintos e não estão relacionados. Ou seja, a classe foi modificada nos sistemas *Source* e *Target*, mas ela não está em conflito direto ou indireto. A Figura 3-7 apresenta um exemplo de um pseudo conflito.

<pre>1 public class A{ 2 int variable =10; 3 4 5 6 7 8 public void methodA(){ 9 print(variable++); 10 } 11 12 public int methodB(int x){ 13 return x*2; 14 } 15 }</pre>	source	<pre>1 public class A{ 2 int variable =10; 3 4 public void methodA(){ 5 print(variable); 6 } 7 8 public int methodB(int x){ 9 return x*x; 10 } 11 12 13 14 15 }</pre>	target
---	---------------	--	---------------

Figura 3-7: Exemplo de Pseudo Conflito

No lado *Source* o método *methodA* foi alterado na classe A, do lado *Target*, a alteração ocorreu no *methodB* da mesma classe. A alteração ocorreu em métodos diferentes de uma mesma classe, como também não existe nenhuma dependência no

grafo de chama entre esses dois métodos. Portanto, eles não apresentam conflitos direto nem indireto, porém como a alteração ocorreu na mesma classe envolvendo até as mesmas linhas de código fonte, sistemas de controle de versão reportarão essas alterações como sendo um conflito, quando na verdade não são.

Os pseudo conflitos são quantificados na abordagem apenas para mensurar o esforço adicional de desenvolvedores ao mesclar sistemas clonados usando ferramentas de *merge* textual. Ferramentas baseadas em análise textual geralmente exibem esses conflitos, mas o uso de ferramentas de *merge* mais avançadas (por exemplo, ferramentas de *merge* estruturado) podem evitar este esforço adicional. Esse tipo de conflito era anteriormente denominado “Conflito Textual” na abordagem proposta por [Lima, et al. 2013]. A Listagem 3-5 apresenta o algoritmo de detecção de pseudo conflitos utilizado pela ferramenta.

Listagem 3-5. Algoritmo de detecção de pseudo conflitos

Para cada classe presente no ChangeLogHistory do sistema *Source*, que **não** está em conflito direto, **nem** em conflito indireto.

Se ela também está presente no ChangeLogHistory do sistema *Target*,
ou seja sofreu uma alteração.

Essa classe apresenta um pseudo conflito

3.4. Estratégia de Resolução de Conflitos

A fase anterior de análise de conflitos serve como guia para a fase de realização do *merge* dos artefatos. A partir dos tipos de conflitos detectados para uma tarefa é possível sugerir uma das três estratégias de resolução do *merge* propostas:

- (i) Merge Automático: quando as alterações podem ser aplicadas sem que sejam adicionados erros ou comportamento inesperados (após o *merge* o código integrado deve se comportar igual a como ele se comportava no sistema *Source*) no sistema *Target* das mudanças. A abordagem sugere que, quando uma determinada tarefa não possuir conflitos ou possuir apenas pseudo conflitos, o *merge* pode ser realizado automaticamente. O exemplo à esquerda na Figura 3-8 , no qual os artefatos distintos e independentes são alterados.
- (ii) Merge Semiautomático: quando as alterações podem ser aplicadas ao

sistema Target, contudo não se tem certeza que não serão incluídos comportamentos inesperados na linha. Como, pela definição, conflitos indiretos são conflitos que não são aplicados ao mesmo atributo ou método, a abordagem sugere que tarefas que possuem apenas conflitos indiretos, podem ser integradas ao sistema *Target*, contudo testes relacionados aos artefatos integrados devem ser executados para se garantir que não foram adicionados erros ao sistema. O exemplo ao centro na Figura 3-8 , no qual artefatos distintos porém que possuem uma dependência sintática entre eles são alterados.

- (iii) Merge Manual: O *merge* de artefatos que possuam conflitos diretos não é contemplado pela abordagem posposta nesse trabalho, devendo o desenvolvedor resolver a integração manualmente. Neste caso, a abordagem ajudará o desenvolvedor fornecendo todos os dados coletados durante a fase de mineração para ajudar o desenvolvedor a realizar o processo de *merge* manual. Nem sempre o desenvolvedor da funcionalidade é a mesma pessoa responsável pela execução do *merge*. Nestes casos, nota-se na prática que quem realiza o *merge* não tem conhecimentos suficientes sobre o que está sendo reconciliado, aumentando as chances de se introduzir erros no sistema. O exemplo à direita da Figura 3-8 , no qual os mesmos artefatos são alterados dos dois lados da evolução.

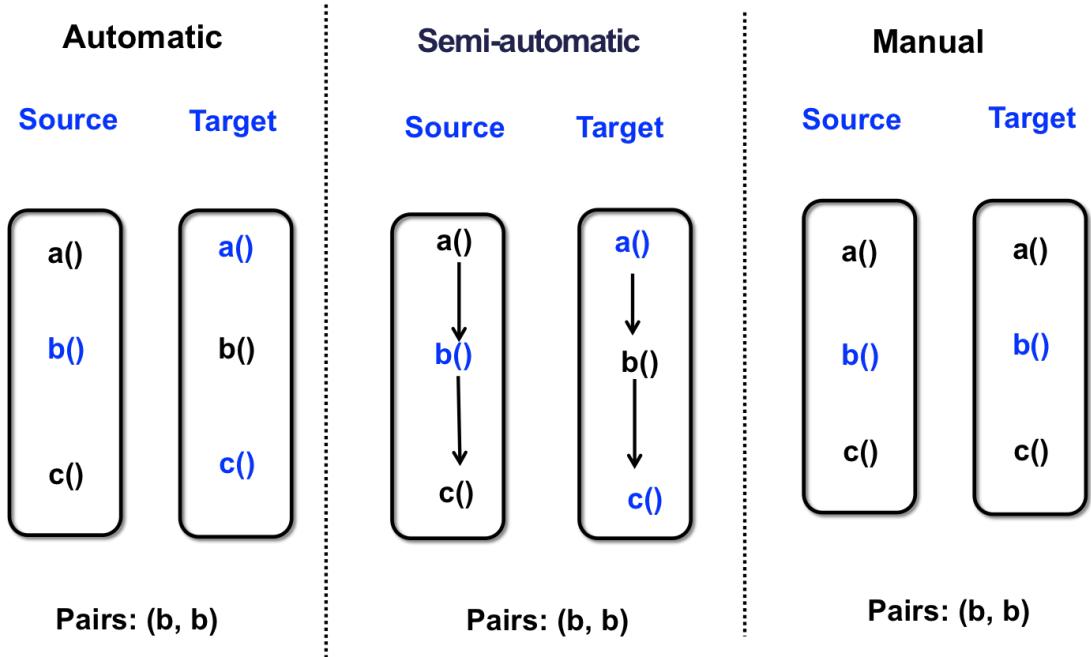


Figura 3-8: Estratégia de Merge com base no tipo de Conflito

Vale ressaltar que o propósito inicial da abordagem não é indicar todos os conflitos que possam surgir no processo de integração. Sabe-se que apenas a indicação de conflitos diretos e conflitos indiretos, com base no grafo de chamada de um artefato, pode não ser o suficiente para identificar todos os tipos de conflitos, ou ainda, podem ser gerados alguns conflitos falsos positivos ou negativos. O objetivo inicial da abordagem é melhorar, em relação as ferramentas de controle de versão atuais, tais como SVN ou GIT, o entendimento sobre a evolução ocorrida, minimizando dos possíveis erros que esse processo custoso está sujeito. A escolha da estratégia de realização de *merge* inicia a próxima fase da abordagem, e é um dos alvos principais da avaliação proposta nesse trabalho.

As informações sobre os conflitos detectados (diretos, indiretos e pseudo conflitos) também são salvas no modelo ChangeLogHistory, e visualizados pelo usuário final da ferramenta, conforme mostrado na Figura 3-9. Essa figura mostra uma árvore estruturada das evoluções ocorridas no sistema *Source*, onde o nó raiz da árvore é a *feature* do sistema em que a evolução ocorreu (caso o mapeamento de variabilidades do sistema seja fornecido). Cada *feature* pode conter várias tarefas e cada tarefa contém as classes Java evoluídas. Os nós filhos da classe são os artefatos de menor granularidade, os métodos e atributos evoluídos. Cada artefato é destacado nas cores vermelha, amarela e

verde dependendo se ele apresenta conflitos diretos, indiretos ou pseudo conflitos respectivamente. Os artefatos “pai” são marcados com o tipo de conflito de maior complexidade para integração entre os conflitos apresentados pelos seus artefatos filhos. Por exemplo, se pelo menos um artefato evoluído em uma tarefa possuir um conflito direto, a tarefa é marcada como apresentando conflito direto.

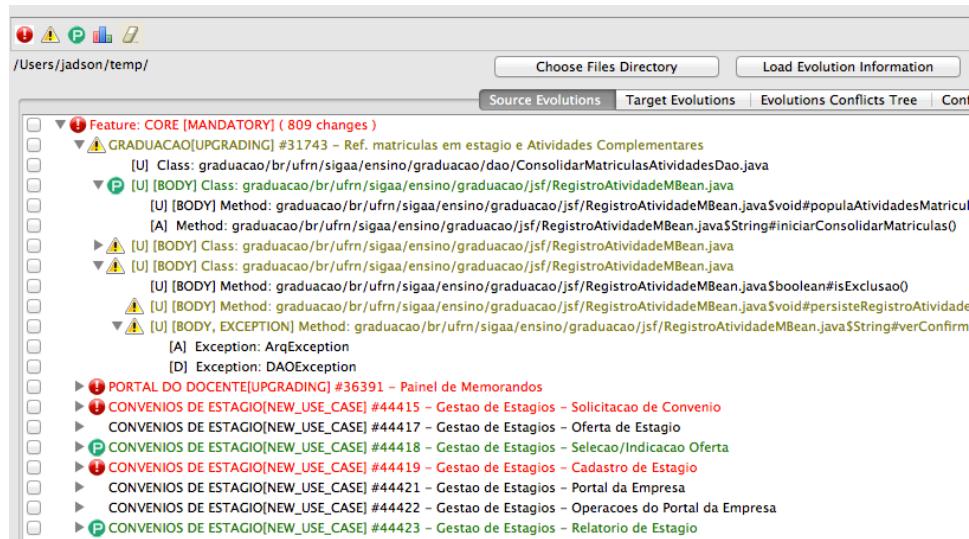


Figura 3-9: Conflitos detectados pelo MergeClear

Outra visualização presente na ferramenta é a árvore de conflitos. Essa visualização mostra todos os artefatos que possuem algum tipo de conflito e todos os artefatos que estão em conflito com ele. Essa visualização foi uma melhoria realizada para esta dissertação e foi útil para a análise qualitativa realizada e pode ser útil também para o usuário visualizar a quantidade exata de conflitos identificados pela ferramenta. A visualização anterior mostra que um artefato está em conflito, mas não com quantos outros artefatos nem quais são esses artefatos. Essa nova visualização é apresentada na *Figura 3-10*.

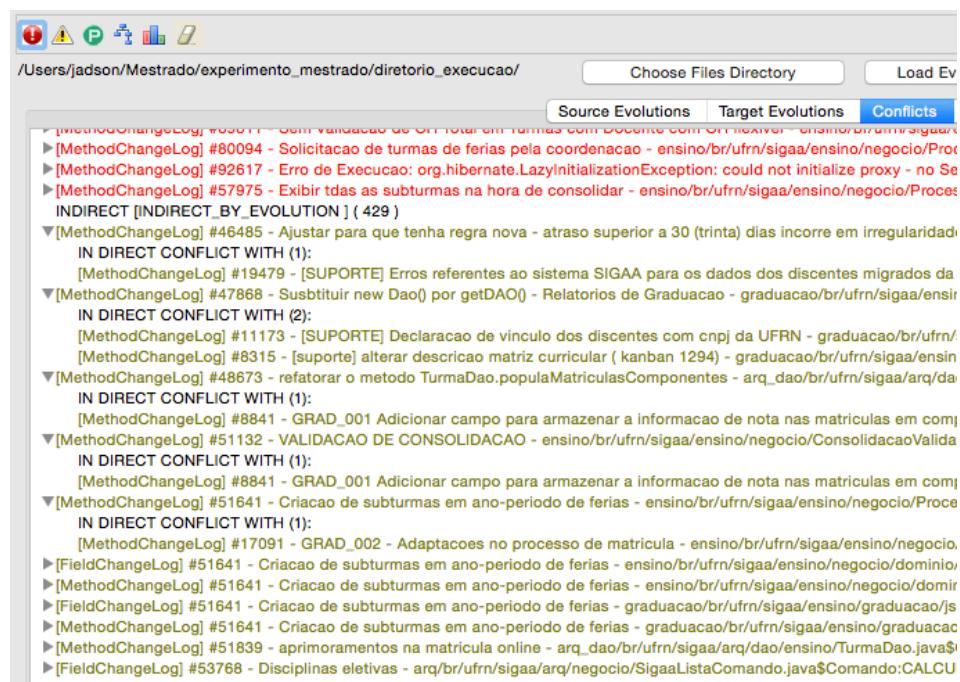


Figura 3-10: Visualização da Árvore e Conflitos da Evolução

3.5. Evoluções Técnicas Realizadas na Abordagem

Esta seção apresenta as principais evoluções técnicas realizadas no *MergeClear* em relação a versão anterior da ferramenta.

3.5.1 Evoluções Técnicas no Cálculo dos Conflitos Indiretos

Uma evolução presente nesta dissertação em relação ao trabalho anterior consiste na geração do grafo de chamadas. Conforme mostrado na Figura 3-11, ao se buscar por referencias de um determinado artefato, de cor vermelha, são retornados os

artefatos que referenciam esse artefato. O *Java Search Engine* não possui por padrão nenhuma busca que retorne os artefatos referenciados a partir do artefato selecionado. Nas versões anteriores da ferramenta, essa mesma busca JDT limitada era realizada nos dois lados da evolução, do lado *Source* e depois do lado *Target*, sendo denominados conflitos de referência e conflitos por dependência respectivamente.

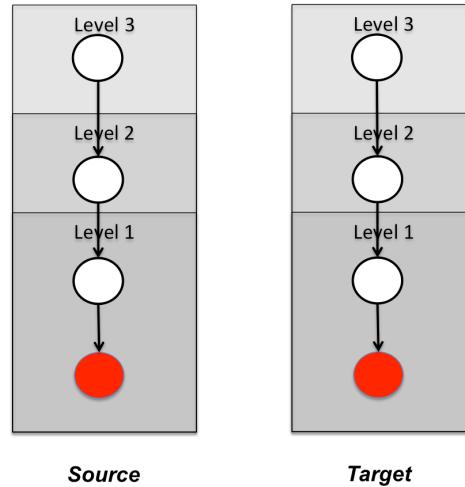


Figura 3-11: Grafo de chamadas usado na versão anterior da ferramenta

Para este trabalho, o algoritmo de montagem do grafo de chamadas foi aprimorado para recuperar o grafo completo de chamadas de um artefato até o nível desejado, tanto dos artefatos que referenciam o artefato selecionado, quanto dos artefatos referenciados a partir do artefato selecionado. Gerando um Grafo Dirigido Acíclico [DIRECTED, 2015] completo conforme mostrado na Figura 3-12.

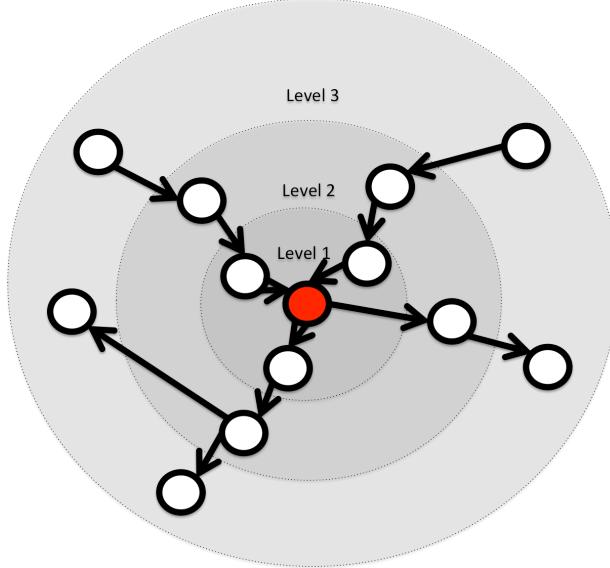


Figura 3-12: Grafo de Chamadas completo entre artefatos usados na análise de conflitos indiretos

Para montar o grafo utilizando a busca JDT foi utilizado o algoritmo descrito na Listagem 3-4.

Listagem 3-4. Algoritmo de geração do grafo Dirigido Acíclico utilizando JDT

Para cada método ou campo X do changeLogHistory do sistema *Source*

Retorne busqueArtefatosReferenciam [X]

Fim algoritmo referenciam

Para cada método ou campo X do changeLogHistory do sistema *Source*

```
/* complementoList = Todos os outros métodos ou campos constantes
 * no changeLogHistory não sejam o artefato X. */
complementoList = changeLogHistory do Source - X
```

```
Para cada elemento N em complementoList
    refsN = busqueArtefatosReferenciam [N]
    if(X percente a refsN){
        // N é referenciado por X
        refsRetorno = refsRetorno + N;
    }
```

Retorne refsRetorno;

Fim algoritmo referenciados

busqueArtefatosReferenciam(X)

refList = Lista de métodos ou campos que referenciam X
utilizando a busca do Java Search Engine

Para cada elemento N em refList até o nível de profundidade
refList = refList + busqueArtefatosReferenciam(refList[N])

Retorne refList

Fim busqueArtefatosReferenciam

O que o algoritmo da Listagem 3-4 faz é usar a lógica inversa. Já que o JDT só possui uma busca que recupera os artefatos que chamam o artefato analisado, a ferramenta verifica para todos os outros artefatos, se tem algum que é chamado a partir do artefato analisado. Essa estratégia é muito mais custosa, só foi possível de ser utilizada porque a ferramenta sempre que busca as referências de um determinado artefato, mantém essa informação em um TreeMap [ORACLE AND/OR ITS AFFILIATES, 2014] em memória. Cujo o custo da operação de recuperação é $\log(n)$. Com o passar do tempo, a maioria das buscas por referências de artefatos tentem e serem recuperadas desse cache em memória, em vez de recuperar a referência do sistema de controle de versão, tornado a montagem do grafo completo de chamadas viável.

Destaca-se que esse grafo de chamadas é executado para o código do lado *Target* da evolução. Ou seja, para saber se determinado artefato do *Source* está em conflito indireto, recupera o artefato equivalente no lado *Target* da evolução e partir desse artefato, recupera-se os artefatos que fazem referência a ele ou que são referenciados a partir dele. Para identificar um artefato como sendo equivalente ao outro, cada artefato no modelo ChangeLogHistory possui um campo assinatura que identifica unicamente um artefato e é utilizado para determinar se um artefato no *Source* é equivalente ao artefato no *Target*, Figura 3-13 mostra as definições das *Strings* usadas para identificar alguns tipos de artefatos suportados pelo modelo.

```

* For features: name[type]
* For change logs: identify[type]
* For classes for example: "br/ufrn/example/A.java"
* For fields: ClassSignature + '$' + type:name
* For methods: ClassSignature + '$' +void#a1(double,float) or void#a1() or int#a1()
* For code pieces :ClassSignature + MethodSignature + '@' +methodName_1[begin,end]
*/
protected String signature;

```

Figura 3-13: Assinatura que identifica unicamente um artfato

Por último, outra melhoria realizada no algoritmo de geração do grafo de chamada foi a desconsideração de referências presentes em comentários do código fonte. Percebeu-se durante a realização do estudo quantitativo que, por padrão, o JDT recupera referências de artefatos que estejam presentes em comentários anotados com a anotação “@see”. Felizmente o próprio mecanismo do Java Search Engine possui um método *SearchMatch#isInsideDocComment()*, que permite identificar e eliminar do grafo de chamadas esse tipo de referência, o que ajudou a reduzir o número de falsos positivos na análise realizada para o estudo. Um exemplo de comentário retornado como referência pela busca JDT é mostrado na *Figura 3-14*.

```

/**
 *
 * @see br.ufrn.arq.negocio.ProcessadorComando#validate(br.ufrn.arq.dominio.Movimento)
 */
@Override
public void validate(Movimento mov) throws NegocioException, ArqException {
    ListaMensagens erros = new ListaMensagens();
    MovimentoCadastro movi = (MovimentoCadastro) mov;

    TermoAutorizacaoPublicacao termoAutorizacao = movi.getObjMovimentado();
    if(termoAutorizacao == null)

```

Figura 3-14: Referências em Comentários Retornadas pelo JDT

Neste caso, se o método *validate(Movimento mov)* sofresse alguma evolução, sem desconsiderar referências em comentários, todos os artefatos que tivessem uma relação com o método abstrato *ProcessadorComando#validate(Movimento)* eram recuperados como uma referência.

3.5.2 Análise de Dependência entre Tarefas

Por último, uma outra contribuição deste trabalho foi adicionar a parte de identificação de dependências entre as tarefas não existente na versão anterior da abordagem. A existência de uma dependência entre duas ou mais tarefas, significa que algum artefato de código foi modificado em mais de uma tarefa e é necessário que modificações anteriores sejam integradas, para que as posteriores sejam. Por exemplo, uma hipotética Tarefa 1 criou um novo método, outra hipotética Tarefa 2, realizou uma alteração nesse método. Não é possível realizar a integração da alteração da Tarefa 2 se o método ainda não existe no sistema *Target*. Para que a integração da Tarefa 2 ocorra é necessário obrigatoriamente a integração da Tarefa 1 primeiro.

A identificação dessas dependências é fundamental para que as tarefas possam ser integradas sem erros e foi requerida para responder a primeira questão de pesquisa desse trabalho. Foi construída também uma visualização que mostra as dependências entre as tarefas evoluídas e quais artefatos do código causaram essas dependências. Essa visualização é apresentada na Figura 3-15.

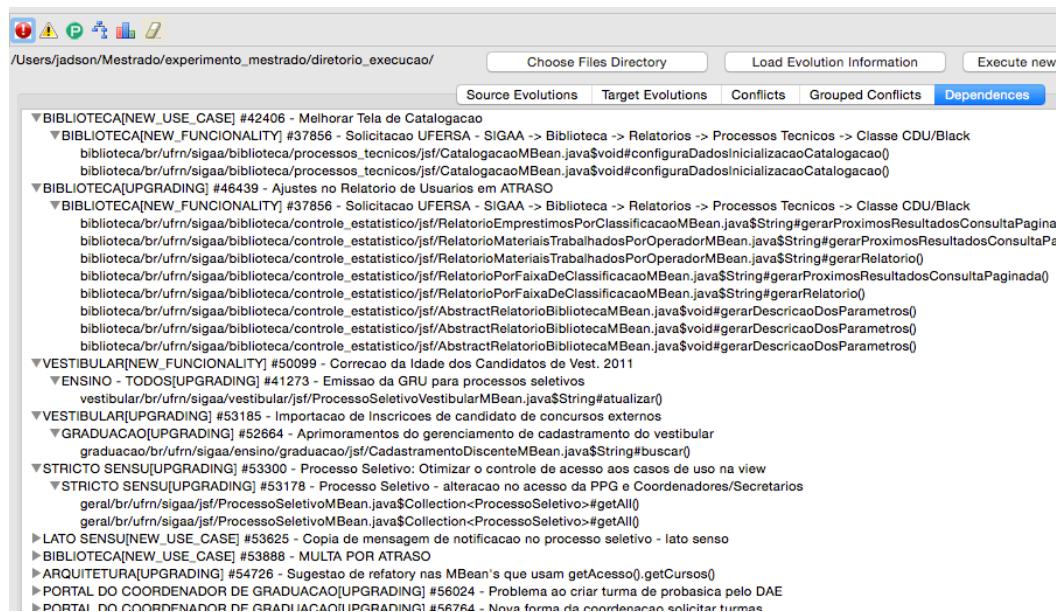


Figura 3-15: Árvore de Dependência entre Tarefas

Na Figura 3-15, a primeira tarefa mostrada de número 42406 depende da tarefa 37856 para ser integrada, e essa dependência foi causada pelo método

configuraDadosInicializacaoCatalogacao() que foi alterado nas duas tarefas entre as versões mineradas para o estudo quantitativo presente nesse trabalho.

A Listagem 3-6 apresenta o algoritmo de detecção de dependências entre tarefas que foi usado nesse estudo.

Listagem 3-6. Algoritmo de Análise de Dependência entre Tarefas

Para cada tarefa, verifique se existe uma tarefa anterior* que alterou o mesmo artefato de código (método ou campo) da tarefa atual.

Se sim, a tarefa atual depende da tarefa anterior.

* Tarefa anterior: Tarefa que foi criada anteriormente à tarefa que está sendo integrada, no sistema de gestão de projetos.

Como será descrito nos resultados do estudo, esse algoritmo ainda não foi suficiente para levantar as dependências de todas as tarefas corretamente. Uma sugestão para melhoria desse algoritmo foi proposta na Seção 4.4.1.1, porém não houve tempo hábil para ser implementada.

4. Estudo Empírico

Este capítulo descreve o estudo empírico realizado com o objetivo de avaliar qualitativamente os resultados obtidos pela ferramenta examinada além de analisar a complexidade de se integrar os conflitos de *merge* identificados. A Seção 4.1 apresenta os objetivos e as questões de pesquisa do estudo conduzido. A Seção 4.2 descreve as características do sistema escolhido para o estudo. A Seção 4.3 apresenta a metodologia aplicada na realização do estudo. A Seção 4.4 discute os resultados obtidos para as três questões de pesquisa e, finalmente, a Seção 4.5 destaca as limitações e ameaças ao estudo.

4.1. Objetivos e Questões de Pesquisa

Após a idealização da abordagem e a construção da ferramenta de *merge* entre sistema clonados, [Lima, 2014] realizou um estudo que quantificou os conflitos encontrados em quatro evoluções clonadas de duas linhas de produtos de software desenvolvidas na linguagem Java. O objetivo do estudo conduzido como parte desta dissertação de mestrado foi realizar uma avaliação qualitativa dos resultados providos por tal ferramenta de forma a validar a qualidade dos resultados gerados pela mesma, além de tentar entender quais tipos de conflitos surgem e qual a complexidade desses conflitos quando se tentar realizar a integração de um sistema que utilizada a abordagem *clone-and-own* para criar variabilidades em seu código.

Um dos pré-requisitos pra a realização desse estudo foi a garantia de que todas as alterações realizadas no código estivessem presentes nas versões analisadas. Em outras palavras, as evoluções paralelas realizadas no código das linhas de produto de software deveriam partir do ponto onde a clonagem foi realizada, possibilitando a utilização pela ferramenta da técnica de Three-Way Merging [T. Mens. 2002] conforme mostrado na Figura 4-1.

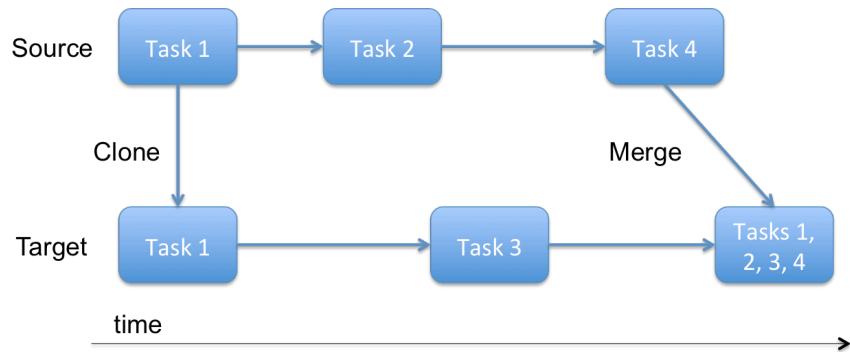


Figura 4-1: Evoluções a partir do ponto de clonagem

Para alcançar os objetivos propostos pelo estudo, foram levantadas três questões de pesquisa:

QP1: Tarefas identificadas pela abordagem como não tendo conflitos podem de fato ser integradas?

Pretende-se com esta questão de pesquisa verificar se as tarefas classificadas pela abordagem como “sem conflitos” realmente podem ser integradas ao código do sistema *Target*, como a abordagem sugere, ou se outros aspectos da evolução influenciam a integração e precisariam também ser analisados.

QP2: Tarefas indicadas com conflitos diretos, realmente representam problemas de integração?

Esta questão de pesquisa investiga se todos os conflitos diretos indicados pela abordagem proposta estão corretos e qual a complexidade da integração desses conflitos.

QP3: Os conflitos indiretos identificados pela abordagem representam conflitos que trazem problemas para a integração?

Esta questão de pesquisa pretende descobrir o quanto a estratégia de identificação de conflitos indiretos da abordagem que utiliza grafo de chamadas é eficaz com relação a identificação de problemas de integração de tarefas.

4.2. Seleção do Sistema para o Estudo

Considerando o contexto apresentado na Capítulo 1, a Superintendência de Informática da Universidade Federal do Rio Grande do Norte (SINFO) desenvolveu um conjunto de sistemas de gestão de informação web criados para automatizar processos de negócios na universidade com foco em aspectos diferentes e complementares, tais como atividades acadêmicas, administração, planejamento e gestão. Esses sistemas começaram a ser implantado em 2006. O estudo concentra-se em um desses sistemas, o SIGAA – Sistema Integrados de Gestão de Atividades Acadêmicas. A Tabela 4-1 apresenta uma visão geral do SIGAA em termos de usuários e tamanho.

Tabela 4-1: Tamanho e quantidade de usuário do SIGAA

Sistema	Total de Usuários	Média de Acessos diários	KLOC	Número de Classes	Número de Métodos
SIGAA	52000	56000	833	5906	81102

O SIGAA foi escolhido porque é um sistema de informação de larga escala, implementado na linguagem Java, e no qual é aplicada da estratégia *clone-and-own* [Rubin, el al. 2012]. Além disso, houve acesso ao seu código fonte e a ferramenta de gerenciamento de configuração e de mudanças utilizada pela Superintendência de Informática da UFRN, correspondente ao sistema *Source* do estudo, assim como ao código fonte e o sistema de gerenciamento de configuração e de mudanças utilizado pela SIG Software e Consultoria, empresa que mantém o sistema *Target*, facilitando assim a mineração das informações necessárias para o estudo.

O SIGAA foi desenvolvido utilizando a linguagem Java e adota uma arquitetura em camadas composta por quatro camadas principais que são: (i) apresentação, uma camada de interface web com o usuário que é codificado usando o framework Java Server Faces; (ii) negócios mantém as regras de negócio associados às funcionalidades do sistema; (iii) persistência responsável por salvar, atualizar, remover ou pesquisar dados de domínio usando o framework Hibernate; e (iv) domínio representa as classes entidades do sistema que armazenam as informações manipuladas.

O cenário escolhido para o estudo foi uma clonagem do sistema SIGAA da Universidade Federal do Rio Grande do Norte (UFRN) para atender regras da Universidade Federal do Oeste do Pará (UFOPA). Do sistema da UFRN foi analisada pela ferramenta a evolução das versões 3.6.0 e 3.8.0, ocorridas entre 15/02/2012 e 13/08/2012, totalizando 1083 tarefas. Já no lado do sistema *Target* da UFOPA, foram analisadas 99 tarefas realizadas entre as versões 3.6.0 e 3.7.64, publicadas entre 06/09/2012 e 26/08/2014, que eram as versões disponíveis a partir do momento em que a clonagem ocorreu.

4.3. Metodologia do Estudo

O estudo consiste em executar a mineração para uma determinada evolução de um sistema web onde a abordagem de *clone-and-own* foi utilizada, e a partir dessa evolução executar a análise de conflitos desenvolvida. A partir dos dados gerados pelas fases de mineração e análise de conflitos, foram analisadas manualmente a integração de uma amostra das tarefas e *commits* indicados pela ferramenta como sendo conflitantes, com o objetivo de medir a qualidade dos resultados da ferramenta que implementa a abordagem.

A partir do momento de clonagem do sistema, várias tarefas foram desenvolvidas no sistema *Source* enquanto o sistema *Target* continuavam evoluindo. Ao tentar integrar uma determinada tarefa, vários problemas podem ocorrer. O presente trabalho avaliou, realizando várias das integrações analisadas pela ferramenta de forma manual, se os dados fornecidos pela ferramenta que implementa a abordagem correspondiam a realidade.

Para realizar esse estudo qualitativo, a ferramenta de merge foi executada para as duas evoluções previamente descritas (SIGAA UFRN 3.6.0 a 3.8.0 e SIGAA UFOPA 3.6.0 e 3.7.64). Na primeira fase da execução foi realizada a mineração dos dados do repositório de código fonte (Subversion) e do sistema de gerenciamento de configuração e de mudanças (Iproject), retornando 1083 tarefas do lado *Source* e 99 do lado do *Target*. Em seguida, foram executadas as três análises de conflitos na sequência: Análise de Conflitos Diretos, Análise de Conflitos Indiretos e Análise de Pseudos Conflitos. Resultado no seguinte número de conflitos:

Tabela 4-2: Resultado da Análise de Conflitos da Evolução Estudada

Tipo de Conflito	Quantidade
Total de Conflitos Diretos	120
Total de Conflitos Indiretos no 1º Nível	328
Total de Conflitos Indiretos no 2º Nível	327
Total de Conflitos Indiretos no 3º Nível	192
Total de Pseudo Conflitos	276
Total de Conflitos	1.243

Para esse trabalho um conflito, tais como os apresentados na Tabela 4-2, representa um par de elementos (atributo ou método) de uma classe do *Source* e do *Target* que estejam em conflitos. No caso de conflitos diretos, um elemento de um artefato só pode estar em conflito direto com ele mesmo. Já no caso de conflitos indiretos, se um elemento de um artefato do *Source* estiver em conflito com outros três elementos de artefatos no grafo de chamadas do *Target*, são contabilizados três conflitos indiretos, um para cada par.

A análise de conflitos indiretos, com a geração do grafo de chamadas entre os artefatos do código fonte do sistema, foi executada em cima do código da versão final do sistema *Target*. Os código dos sistemas utilizados para o estudo foram versões finalizadas que representam *branches* específicos do sistema de controle de versão. A Figura 4-2 mostra o arquivo de configuração da ferramenta com as configurações utilizadas para esse estudo.

```

# SOURCE SYSTEM
SOURCE_SYSTEM_URL=http://localhost:8080/iproject/public/rest/buildevolutioninformation
SOURCE_SYSTEM_USER=
SOURCE_SYSTEM_PASSWORD=234530cbf9f28b29ce71edca2fb84399128dde94
SOURCE_SYSTEM_NAME=SIGAA
SOURCE_MODULE_NAME=
SOURCE_SYSTEM_START_VERSION=3.6.0
SOURCE_SYSTEM_END_VERSION=3.8.0
SOURCE_SYSTEM_OFF_LINE_EVOLUTION_FILE=buildInformation_SIGAA_3.6.0_3.8.0.xml

# SOURCE REPOSITORY
# TAKE THE CODE TO ANALYZE FROM A STABLE BUILD
SOURCE_REPOSITORY_URL=http://desenvolvimento.info.ufrn.br/projetos
SOURCE_REPOSITORY_PATH=/tags/SIGAA_3.8.0
SOURCE_REPOSITORY_USER=jadson
SOURCE_REPOSITORY_PASSWORD=

# TARGET SYSTEM
TARGET_SYSTEM_URL=http://localhost:8080/iproject/public/rest/buildevolutioninformation
TARGET_SYSTEM_USER=
TARGET_SYSTEM_PASSWORD=234530cbf9f28b29ce71edca2fb84399128dde94
TARGET_SYSTEM_NAME=SIGAA
TARGET_MODULE_NAME=
TARGET_SYSTEM_START_VERSION=3.6.4
TARGET_SYSTEM_END_VERSION=3.7.64
TARGET_SYSTEM_OFF_LINE_EVOLUTION_FILE=buildInformation_SIGAA_3.6.4-3.7.64-ufopa.xml

# TARGET REPOSITORY
TARGET_REPOSITORY_URL=http://version.info.ufrn.br/cooperacao
TARGET_REPOSITORY_PATH=/branches/sigref/tags/sigref/PROJETOS/UFOPA/PRODUCAO/SIGAA_3.7.64/SIGAA
TARGET_REPOSITORY_USER=jadson
TARGET_REPOSITORY_PASSWORD=

```

Figura 4-2: Configurações da Ferramenta de Merge utilizadas para o estudo

Após a análise de conflitos as tarefas foram agrupadas de acordo com alguns critérios de complexidade. Os critérios principais de agrupamentos usados foram tipo da tarefa e tipo de conflito detectado, os quais são detalhados a seguir.

O tipo da tarefa:

- i. NEW_FUNCTIONALITY – Tarefas criadas para implementação de funcionalidade totalmente novas.
- ii. UPGRADING – Tarefas criadas para implementação de aprimoramentos em funcionalidades já existentes.
- iii. BUG_FIX – Tarefas criadas exclusivamente para correção de erros, sem adicionar nenhuma nova funcionalidade ao sistema.

Tipo de conflito detectado:

- i. No Conflicts: Não foi detectado nenhum tipo de conflito nos artefatos da tarefa.

- ii. Direct Conflicts: A tarefa apresentou conflitos diretos.
- iii. Indirect Conflicts: A tarefa não apresentou conflito direto mas apresentou conflitos indiretos.
- iv. Pseudo Conflicts: A tarefa não apresentou conflitos diretos nem indiretos, mas apresentou pseudo conflitos.

As tabelas Tabela 4-3 e Tabela 4-4 apresentam a quantidade de tarefas resultante para esses critérios de agrupamento.

Tabela 4-3: Classificação das Tarefas da Evolução do Sistema Source

	No	Direct	InDirect	Pseudo	Total
NEW_FUNCTIONALITY	142	10	50	20	222
UPGRADING TASKS	212	25	77	39	353
BUG_FIX TASKS	352	26	77	53	508
TOTAL	706	61	204	112	1083

Tabela 4-4: Classificação das Tarefas da Evolução do Sistema Target

	No	Direct	InDirect	Pseudo	
NEW_FUNCTIONALITY	34	25	21	2	82
UPGRADING TASKS	3	5	7	1	16
BUG_FIX TASKS	0	0	1	0	1
TOTAL	37	30	29	3	99

E por último, as tarefas foram agrupadas utilizando mais dois critérios: (i) o tamanho da tarefa, isto é, quantidade de artefatos alterados na tarefa; e (ii) se os artefatos alterados pertenciam ao mesmo subsistema ou a tarefa alterava artefatos pertencentes a mais de um subsistema. A Tabela 4-5 e a Tabela 4-6 apresentam a média de classes alteradas em cada agrupamento realizado anteriormente.

Tabela 4-5: Médias de classes Java alteradas por tipo de tarefa e tipo de conflito no Source

	No Conflits	Direct Conflits	InDirect Conflits	Pseudo Conflits
NEW_ FUNCTIONALITY TASKS	8.5	8.9	8.5	7.95
UPGRADING TASK	7.48	14.88	20.04	6.41
BUG_FIX TASKS	2.01	1.80	3.97	1.52

Tabela 4-6: Média de classes Java alteradas por tipo de tarefas e tipo de conflito no Target

	No Conflits	Direct Conflits	InDirect Conflits	Pseudo Conflits
NEW_ FUNCTIONALITY TASKS	1.38	3.92	2.66	1.66
UPGRADING TASK	1.5	4.6	3.0	11
BUG_FIX TASKS	0	0	1	0

A partir dessa classificação, foram selecionadas tarefas, atendendo aos requisitos de cada questão de pesquisa, buscando representar diferentes níveis de esforço de integração. Por exemplo, foram selecionadas desde tarefas de correção de erros, que tiveram poucas classes alteradas dentro de um mesmo módulo do sistema, até tarefas de novas funcionalidades que tiveram uma quantidade considerável de classes alteradas, em vários módulos do sistema. As tarefas selecionadas para o estudo estão dispostas no Apêndice I dessa dissertação.

4.4. Resultados do Estudo Empírico

Esta seção apresenta e discute os resultados do estudo empírico para cada uma das questões de pesquisa definidas anteriormente.

4.4.1 QP1: Tarefas identificadas pela abordagem como não tendo conflitos podem de fato ser integradas?

O objetivo da primeira questão de pesquisa foi avaliar se realmente as tarefas que a abordagem classifica como não possuindo conflitos, podem ser integradas de forma automática, sem gerar novos problemas de compilação no sistema *Target* após a integração. Ou seja, a estratégia de se identificar conflitos de código com base em grafo de chamadas dos artefatos de código é suficiente para garantir que a integração dos trechos alterados no *Source* não adicionem erro de compilação ao sistema *Target*?

Para esse trabalho é considerada integração sem erro quando uma cópia do código alterado no sistema *Source* é movido diretamente para o *Target* mas isso não ocasiona erros de compilação. Por dificuldades técnicas é difícil obter a versão exata do banco de dados que possibilitasse executar o sistema, por isso não foi possível executar a aplicação e constatar que não existem erros lógicos no sistema. O sistema também não possui testes automatizados que poderiam ser executados para auxiliar a descoberta de erros de execução.

Para responder essa questão de pesquisa foi aplicada a metodologia descrita na Seção 4.3, que envolveu executar a mineração, a análise de conflitos e agrupar tarefas com características semelhantes.

Após esses passos, foram selecionadas tarefas identificadas pela abordagem como não possuindo nenhum tipo de conflito. A seleção foi feita com base no conhecimento do sistema, tentando-se escolher tarefas que representassem diferentes cenários de integração.

Foram analisadas várias tarefas durante a evolução da ferramenta, a cada vez que se constatava resultados não satisfatórios ou erros da ferramenta, uma correção era feita e as novas tarefas eram descartadas. Para responder a QP1, na versão final da ferramenta, foram selecionadas e analisadas 15 tarefas. A lista completa de tarefas

utilizadas para responder essa questão de pesquisa encontram-se no Apêndice I desta dissertação

Um fator percebido nas primeiras análises qualitativas realizadas foi que apenas o critério de conflitos entre artefatos não era suficiente para garantir que a integração ocorresse sem adicionar erros de compilação no sistema *Target*. Era preciso também analisar as dependências entre tarefas. Uma tarefa só pode ser integrada sem erros de compilação ao *Target*, se e somente se, ela não possuir conflitos e todas as tarefas que ela depende, também não possuam conflitos e sejam integradas antes dela.

Assim, foi implementada uma nova análise automatizada na abordagem. Além da análise de conflitos, a abordagem para realização de *merge* entre sistema evoluídos em paralelo deve também calcular a dependência entre as tarefas que estão sendo integradas. A Figura 4-3 apresenta uma parte da árvore de dependências entre as tarefas que estão sendo reportadas nesse estudo.

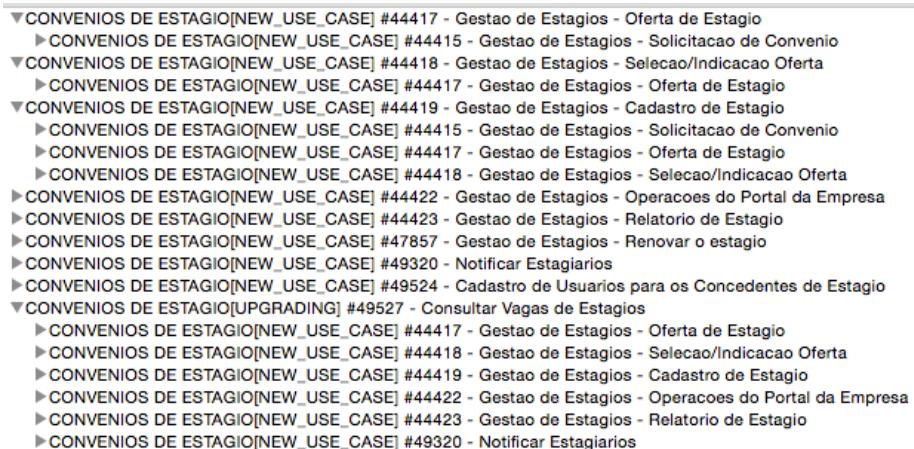


Figura 4-3: Dependências entre Tarefas

A Figura 4-3 mostra que a tarefa 44419 depende que as tarefas 44415, 44417 e 44418 sejam integradas primeiro, por sua vez, a tarefa 44418 depende da 44417 que depende da tarefa 44415, por isso várias desses tarefas estão repetidas. Todo essa dependência deve ser analisada recursivamente para possibilitar que a integração ocorra sem erros.

4.4.1.1 Resultado da análise da primeira questão de pesquisa

A Tabela 4-7 apresenta os resultados da análise qualitativa realizada para responder a primeira questão de pesquisa. Das 15 tarefas analisadas do sistema *Source* que não apresentavam conflitos, 11 tarefas, aproximadamente 73%, puderam ser integradas sem gerar erros de compilação no sistema *Target*, usando a estratégia de Merge automático sugerido pela abordagem.

Tabela 4-7: Quantitativo das tarefas analisadas

Total Tarefas Analisadas	Integração Possível	Integração Possível mas Erro no cálculo das Dependências	Erro durante a fase Mineração	Impossível de ser realizado a análise manual
15 (100%)	11 (73,33%)	1 (6,6%)	1 (6,6%)	2 (13,33%)

Tarefas com Erro no Cálculo de Dependências. Uma das tarefas analisadas apresentou um erro no cálculo da dependências entre tarefas. Na análise da tarefa 31743 - *Ref. matriculas em estagio e Atividades Complementares* não foi detectado que ela dependia da tarefa #89452 - *Mensagem de Erro em Consolidação de Atividade*, visto que o algoritmo implementado utiliza apenas a data de criação da tarefa para determinar as dependências entre elas. Com isso, na realização do *merge* manual sem considerar as mudanças da tarefa 89452, um erro de compilação foi gerado no código do *Target*. Ao se considerar as alterações realizadas na tarefa 89452, o erro de compilação não era gerado, possibilitando a integração correta da tarefa.

O algoritmo de cálculo de dependências apresentado na Listagem 3-6 não é suficiente para determinar se uma tarefa foi implementada antes de outra e precisa também ser integrada primeiro. Em alguns casos é difícil determinar que tarefa foi realizada antes de outra. Isso acontece porque em alguns casos as tarefas ocorrem paralelamente, possuindo *commits* intercalados. Nestes casos um algoritmo que poderia ser aplicado em substituição ao algoritmo da Listagem 3-6 é o algoritmo apresentado na Listagem 4.1. Infelizmente não houver tempo hábil para a sua aplicação nesse estudo.

Listagem 4-1. Algoritmo sugerido de Análise de Dependência entre Tarefas

Para cada commit realizada na tarefa

 Para cada artefato de código presente no commit

 Verificar todos os commits anteriores até o ponto de clonagem

 Se existir algum commit que altere o mesmo artefato e que pertence a outra tarefa

 Tarefas com dependências

 Se não

 Tarefas sem dependência

A Tabela 4-8 apresenta a quantidade de tarefas analisadas que possuíam dependências entre as 15 selecionadas para essa questão de pesquisa e dessas dependências quais foram calculadas corretamente, e quais delas percebeu-se que determinadas dependências deixaram de ser detectadas, através de uma análise manual. Uma aspecto notado durante o estudo é que a dependência entre tarefas é parcial. Isso significa que nem sempre todo o código da tarefa dependente precisa ser integrado para a tarefa desejada poder ser integrada, por exemplo, as vezes a dependência entre duas tarefas se restringe apenas a um método entre um conjunto de artefatos alterados. Aplicando-se apenas a alteração anterior desse método, a dependência já era solucionada permitindo a integração.

Tabela 4-8: Quantitativo das dependências entre tarefas

Total de Tarefas com dependência, entre as selecionadas para a QP1	Tarefas com Dependências Calculadas Corretamente	Tarefas com erro no cálculo das dependências
3	1	2

Tarefa com Erro na Mineração. Outra tarefa presente na Tabela 4-7 apresentou um erro no processo de mineração. Ao realizar a integração manual, verificou-se que a integração não poderia ser realizada, pois a tarefa possuía um conflito direto não detectado pela ferramenta. Ao analisar esse problema percebeu-se que o erro não foi da mineração realizada pela ferramenta e sim de um erro no processo de desenvolvimento. O código do sistema *Target* sofreu uma evolução, porém essa evolução não foi registrada em nenhuma tarefa no sistema de gerenciamento de configuração e de mudanças utilizado pelo sistema *Target*. Com isso, a evolução não foi recuperada e o conflito direto não pôde ser detectado.

Isso mostra que a parte da abordagem que realiza a mineração dos sistemas de configuração e de mudanças é sujeita a erros caso as informações geradas durante o processo de desenvolvimento não sejam documentadas de forma correta nesses sistemas. Caso uma evolução realizada na linha de produtos não seja incluída na respectiva tarefa no sistema de configuração e de mudança utilizado, essa informação é perdida.

Tarefas Descartadas. Por fim, em duas tarefas selecionadas não foi possível realizar nem a integração manual. A tarefa “68883 - NEE -> Relatorios/Consultas -> Consultas Gerais -> Orientacao de Atividades” , apresentou também um erro no processo de desenvolvimento, algumas classes que foram alteradas para implementar as mudanças da tarefa analisada, foram registradas no sistema de gerenciamento de mudanças como pertencendo a outra tarefa, junto com as alterações também realizada nessa outra tarefa. Dessa forma, ficou bastante confuso separar manualmente qual código pertencia a qual tarefa. Por causa disso, ela foi descartada. É um pré-requisito da abordagem que todas as informações estejam corretamente cadastradas no sistema de gerência de mudanças utilizado pelo sistema analisado.

A tarefa “39788 - ENVIAR E-MAIL AOS ALUNOS AVISANDO NOVA NOTICIA NO CURSO” recebeu os primeiro *commits* em 2010, sendo o código *commitado* de forma incompleta no sistema de controle de versão, e somente em 2012, a tarefa foi finalizada por questões de prioridade. Nesse meio tempo, ocorreram vários *commits* que alteraram muito o código, impossibilitando assim fazer a análise manual da tarefa, por isso ela também foi descartada para esse estudo.

Tarefas Integradas Corretamente. As 15 tarefas analisadas para essa questão de pesquisa resultaram em 98 alterações em artefatos de código (campos ou métodos). Para cada um desses 98 artefatos foi verificado manualmente que alterações foram realizadas nesse artefato.

Para cada alteração tentou-se aplicá-la ao código do sistema *Target* verificando-se os possíveis erros de compilação que isso geraria ao sistema *Target*. A Figura 4-4 apresenta o *workspace* onde foi aplicado o estudo depois da integração das 12 tarefas que puderam ser integradas. Como mostrado na Figura 4-4, o *workspace* não apresenta nenhum erro de compilação.



Figura 4-4: Workspace onde foi realizado a integração

Por último, a *Figura 4-5* apresenta parcialmente as classes que foram alteradas durante a aplicação do estudo no sistema *Target*.

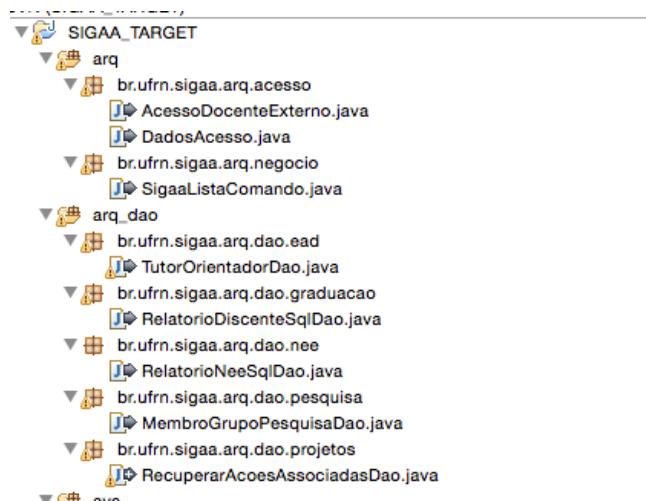


Figura 4-5: Classes alteradas durante a integração manual para responder a QP1

4.4.2 QP2: Tarefas indicadas com conflitos diretos, realmente representam problemas de integração?

O objetivo desta questão de pesquisa foi verificar se os conflitos diretos que a ferramenta detecta são realmente conflitos e se isso é condição suficiente para afirmar

que tais tarefas não podem ser integradas. Para essa questão foram escolhidas algumas tarefas que apresentavam conflitos diretos. O critério para escolha dessas tarefas foi buscar selecionar tarefas que representassem diferentes cenários de integração. Foram selecionada 10 tarefas. A lista completa de tarefas utilizadas para responder essa questão de pesquisa encontram-se no Apêndice I desta dissertação.

De maneira geral todas as tarefas selecionadas realmente apresentavam conflitos diretos, conforme indicado pela abordagem. Ou seja, a ferramenta está identificando de forma correta se um mesmo artefato de código foi alterado nos sistemas *Source* e *Target*.

O segundo objetivo dessa questão de pesquisa é investigar se ter um conflito direto é condição suficiente para afirmar que a tarefa não pode ser integrada. A análise das tarefas do estudo verificou que algumas tarefas mesmo apresentando conflitos diretos poderiam ser integradas, pois muitas vezes alterações no mesmo artefato de código são em linhas de código separadas e não estão diretamente relacionadas ou poderiam coexistir. A Tabela 4-9 apresenta a quantidade de tarefas com conflito diretos analisadas e a quantidade de tarefas que, mesmo ainda possuindo conflitos diretos, existe a possibilidade de integração.

Tabela 4-9: Análise dos Conflitos diretos

Total de Tarefa	Conflitos Diretos com Possibilidade de Integração	Conflitos Diretos sem Possibilidade de Integração
10	6 (60%)	4 (40%)

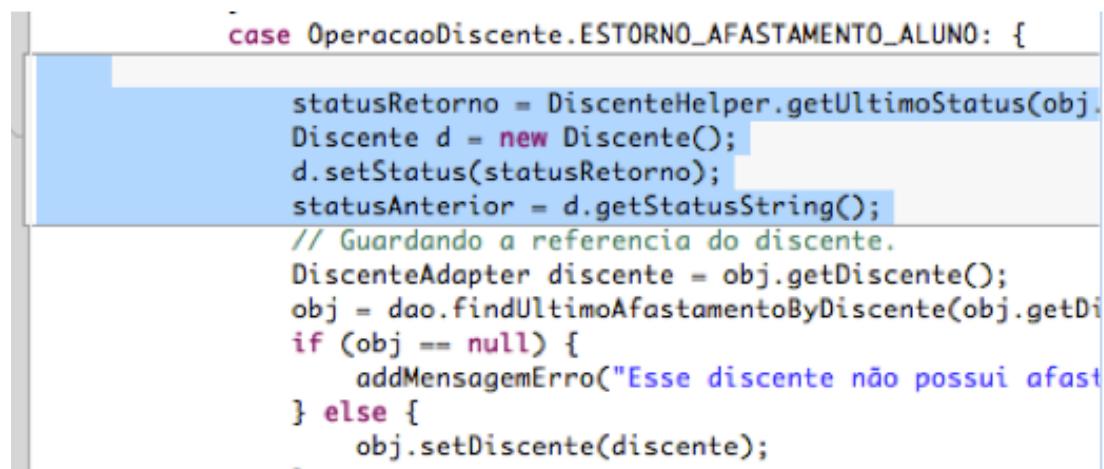
A Tabela 4-9 mostra que 60% das tarefas ainda poderiam ser integradas, mesmo possuindo conflitos diretos corretamente identificados pela ferramenta. Ressalta-se que esses tarefas com possibilidade de integração não representam um erro na ferramenta, pois é necessária uma análise semântica para identificar se existe essa possibilidade de integração e esse não é o objetivo da abordagem. As seções 4.4.2.1 e 4.4.2.2 apresentam respectivamente exemplo de análise de tarefas com conflitos diretos que ainda permitem integração e tarefas que mesmo com a análise manual não há possibilidade de integração.

Além dos resultados anteriormente descritos, destaca-se o resultado da análise da Tarefa 47140 - *Nomear Grupos*. Essa tarefa apresentou conflitos para todos os seus artefatos, isso ocorreu porque as mudanças realizadas nessa tarefa já haviam sido integradas ao código no *Target* analisado, SIGAA UFOPA 3.7.64, por isso foi gerado esse resultado fora do padrão. Isso ocorre porque a análise de conflitos diretos é sempre realizada com a última versão do código da evolução. Algumas tarefas podem ter sido integradas em uma das várias versões intermediárias. Até o momento a abordagem não guarda nenhum tipo de informação sobre tarefas que já foram integradas. Para esse caso, ainda foi realizada uma verificação para saber se o código presente no *Target* correspondia à evolução identificada pela ferramenta no *Source*, para reforçar que, se o código não tivesse sido integrado, a tarefa não apresentaria conflitos diretos e poderia ser integrada, como realmente foi.

4.4.2.1 Tarefas com conflitos diretos que podem ser solucionados

Tarefa 95931 - Excluir trancamento de programa indevido

A tarefa de número 95931 apresentou um conflito direto no método *MovimentacaoAlunoMBean#selecionarDiscente()*. No lado do *Source* ocorreu uma alteração no status que o discente recebe em caso de estorno do afastamento, como mostrado na Figura 4-6.



```
case OperacaoDiscente.ESTORNO_AFASTAMENTO_ALUNO: {
    statusRetorno = DiscenteHelper.getUltimoStatus(obj);
    Discente d = new Discente();
    d.setStatus(statusRetorno);
    statusAnterior = d.getStatusString();
    // Guardando a referencia do discente.
    DiscenteAdapter discente = obj.getDiscente();
    obj = dao.findUltimoAfastamentoByDiscente(obj.getDi
    if (obj == null) {
        addMensagemErro("Esse discente não possui afast
    } else {
        obj.setDiscente(discente);
    }
}
```

Figura 4-6: Mudança da tarefa 87467 no Source

Por outro lado, no *Target* a alteração foi realizada para o cancelamento e conclusão do aluno, Figura 4-7. Apesar de estarem no mesmo artefato de código (mesmo método), tais mudanças foram realizadas em trechos de código diferentes e as operações de cancelamento e conclusão ocorrem em momento diferentes da vida acadêmica do aluno que a operação de afastamento. Para quem conhece as regras de negócio do sistema analisado, claramente essas evoluções poderiam ser integradas. Uma forma de permitir que tais tarefas fossem integradas facilmente e não apresentassem conflitos seria que tal código fosse refatorado para isolar tais comportamentos independentes através da sua extração para métodos diferentes. Nesse caso, tal conflito direto seria transformado em um conflito indireto.

```

case OperacaoDiscente.CANCELAR_PROGRAMA: {
    // Validar status válidos para cancelamento de programa
    List<Integer> statusValidos = Arrays.asList(StatusDiscente.ATIVO,
        StatusDiscente.SUSPENSO);
    if (!statusValidos.contains( obj.getDiscente().getStatus() ) )
        addMensagemErro("O discente escolhido não pode ter seu progra
    }
    break;
}
case OperacaoDiscente.CONCLUIR_ALUNO: {
    /*UFOPA - Alteração Específica*/
    if(obj.getDiscente().getCurso().getTipoDiplomacao().getId() == Tip
        addMensagemErro("Não é possível concluir um aluno de um curs
    }
    /*UFOPA - Fim Alteração Específica*/
}

```

Figura 4-7: Mudança da tarefa 87467 no Target

Tarefa 67401 - Solicitacoes Lato Sensu

Essa tarefa apresenta um conflito direto no método *LogonBean#execute*. Como pode ser visto na Figura 4-8, a mudança realizada no *Source* para essa tarefa foi a adição de um novo papel, para permitir a usuários que possuam esse papel se *logarem* no sistema. A variável *administradores* que armazena uma lista de permissões que permitem realizar o *login* no sistema foi modificada para adicionar uma nova permissão.

```

UsuarioMov userMov = (UsuarioMov) mov;

if (userMov.getCodMovimento().getId() == ArqListaComando.LOGAR_COMO_COD )
    int[] administradores = { SigaaPapeis.ADMINISTRADOR_SIGAA, SigaaPapeis
        checkRole(administradores, mov);
    }

Usuario user = userMov.getUsuario();
UsuarioDao dao = getDAO(UsuarioDao.class, mov);
UsuarioDao daoUpdate = null;

try {
    ...
}

```

Figura 4-8: Adição de mais um papel para realizar o login no sistema

A Figura 4-9 mostra a alteração realizada no *Target* que gerou o conflito direto identificado pela ferramenta. A alteração realizada diz respeito à funcionalidade “nomeSocial”. Ela está presente em trechos de código distintos e não impacta em nada na permissão de realizar o *logon* realizada no *Source*. Então essas mudanças poderiam ser integradas. A mesma abordagem de extrair o código de tais funcionalidades para métodos diferentes sugerido para o outro exemplo apresentado nesta seção poderia ser utilizado nesse caso, para evitar tal conflito direto. Nota-se que muitos dos conflitos diretos gerados são oriundos do alto acoplamento do código fonte do sistema analisado.

```

USUARIO usuario = usuariotemporaryKey(userMov.getUsuario()).getUsua
String nomeSocial = usuario.getPessoa().getNomeSocial();
usuario.getPessoa().setNomeSocial("");
/** Customização UFOPA - SIGProject 19000 */
String nomeSocial = usuario.getPessoa().getNomeSocial();
usuario.getPessoa().setNomeSocial("");
/** Customização UFOPA - SIGProject 19000 */

```

Figura 4-9: Alteração realizada no Target

4.4.2.2 Tarefas com conflitos diretos que necessitam de análise da semântica para a integração

Tarefa 88512 - 009 - Matricula de ingressantes

A tarefa 88512 possui um exemplo de conflito direto de difícil resolução, mesmo realizado uma análise manual. Como pode ser visualizado na Figura 4-10 e na

Figura 4-11, as alterações nesse método (marcadas na cor cinza) foram feitas nas mesmas linhas e envolvendo as mesmas regras de negócio. É muito difícil fazer uma integração nesse caso, mesmo que manual. O desenvolvedor deve conhecer muito bem as regras de negócio envolvidas nesse caso de uso para fazer as duas alterações coexistirem. Ou então escolher uma delas e usar como código integrado, descartando a outra.

```
for (Entry<Integer, List<MatriculaEmProcessamento>> entry : matriculas.entrySet()) {
    int ordem = 0;
    int vagasIngressantes = 0;

    for (MatriculaEmProcessamento matricula : entry.getValue()) { // Verifica se a matrícula pode ser processada

        ReservaCurso reserva = vagasReservadasPorMatriz.get(new MatrizCursoKey(
            entry.getKey(), matricula.getMatricula().getAnoLetivo(),
            matricula.getMatricula().getPeriodoLetivo()));

        if (reserva.getTotalVagasReservadas() > 0 && capacidade > 0) {
            if (reserva.getVagasReservadasIngressantes() > 0 && matricula.getMatricula().getSituacao() == SituacaoMatricula.SUSPENSAO) {
                reserva.setVagasReservadasIngressantes((short) (reserva.getVagasReservadasIngressantes() - 1));
            } else if (reserva.getVagasReservadas() > 0) {
                reserva.setVagasReservadas((short) (reserva.getVagasReservadas() - 1));
            } else {
                sobras.add(matricula);
                break;
            }
            capacidade--;
            matricula.setSituacao(SituacaoMatricula.MATRICULADO);
        }
    }
}
```

Figura 4-10: Alterações do método processar no Source

```

private ProcessamentoMatriculasTurma() { }
/*UFOPA - Alteração Específica - Tarefa 17091*/
public List<MatriculaEmProcessamento> processar() throws DAOException {
    List<MatriculaEmProcessamento> matriculasAposProcessamento = new ArrayList<MatriculaEmProcessamento>();
    List<MatriculaEmProcessamento> sobras = matriculas.get(0); // Matrículas a serem consolidadas
    matriculas.remove(0);
    ProcessamentoMatriculaGraduacaoUFOPADao dao = DAOFactory.getInstance().getDAO(ProcessamentoMatriculaGraduacaoUFOPADao.class);
    for (Entry<Integer, List<MatriculaEmProcessamento>> entry : matriculas.entrySet()) {
        int ordem = ValidatorUtil.isEmpty(entry.getValue()) || ValidatorUtil.isNaN(entry.getValue().size()) ? 0 : entry.getValue().size();
        for (MatriculaEmProcessamento matricula : entry.getValue()) {
            ReservaCurso reserva = vagasReservadasPorMatriz.get(new MatrizCurso(matricula));
            if (reserva.getVagasReservadas() > 0 && capacidade > 0 && !possuiNaoReservada(reserva)) {
                reserva.setVagasReservadas((short) (reserva.getVagasReservadas() + 1));
                capacidade--;
            }
        }
    }
}

```

Figura 4-11: Alterações do método *processar* no *Target*

88001 - Turma não consolida quando já existe uma matrícula consolidada

Nessa tarefa ocorreu um conflito direto no método *ProcessadorCadastroNotasUnidades#execute()*. Realizando a análise manual, percebe-se que, foi realizada uma alteração no *Target* para corrigir o mesmo erro da consolidação de turmas já realizada no *Source*. Contudo, a correção do lado do *Target* foi realizada de uma forma diferente que a correção que foi realizada no *Source*, deixando claro que não se tratou de uma integração previamente realizada.

Inclusive uma validação implementada no *Source* não está presente do *Target*. Ou seja, ou a implementação do *Source* ou a do *Target* está incompleta. A Figura 4-12 e a Figura 4-13 mostram essas alterações. Como claramente as alterações foram realizadas para corrigir um erro na mesma funcionalidade do sistema e ocorreram nas mesmas linhas do código fonte, elas não podem ser integradas, representando de fato um conflito direto cuja integração requer uma análise semântica.

```

List<Integer> listaNumeroUnidades = cadastroNUDao.countNotaUnidadeByTu
// Se a turma não estiver aberta ou se possuir matrículas consolidadas
// Caso não exista unidades na turma e exista matrículas consolidadas
if (!turma.isAberta() || (cadastroNUDao.existeMatriculasConsolidadas(turma)
    return matriculas;

Integer numUnidades = TurmaUtil.getNumUnidadesDisciplina(turma);

List<MatriculaComponente> notasARemover = new ArrayList<MatriculaCompon
List<MatriculaComponente> notasACadastrar = new ArrayList<MatriculaCom

// Se a lista contém mais de 1 elemento é porque na turma existem alunas
// Se a lista não contém o número de unidades corretos é porque todos
if (listaNumeroUnidades.size() > 1 || !listaNumeroUnidades.contains(numUnidades))
    for (MatriculaComponente matricula : matriculas) {

        if (!matricula.isConsolidada() && matricula.getNotas().size() < numUnidades)
            notasARemover.add(matricula);
        else if (matricula.getNotas().size() < numUnidades)
            notasACadastrar.add(matricula);
    }
}

```

Figura 4-12: Correção do erro da consolidação no lado do Source

```

public Object execute(Movimento mov) throws NegocioException, ArqException {
    CadastroNotasUnidadesDao cadastroNUDao = getDAO(CadastroNotasUnidadesDao.class);
    matriculaReferencia = null;

    try {
        MovimentoCadastro cMov = (MovimentoCadastro) mov;
        Turma turma = (Turma) cMov.getObjMovimentado();
        @SuppressWarnings("unchecked")
        Collection<MatriculaComponente> matriculas = (Collection<MatriculaCompon

        // Se a turma não estiver aberta ou se possuir matrículas consolidadas
        if (!turma.isAberta()
            || existeMatriculasConsolidadas(matriculas))
            return matriculas;

        List<Integer> listaNumeroUnidades = cadastroNUDao.countNotaUnidadeByTu
    }
}

```

Figura 4-13: Correção do erro da consolidação no lado do Target

4.4.3 QP3: Os conflitos indiretos identificados pela abordagem representam conflitos que trazem problemas para a integração?

Como última questão de pesquisa, este trabalho analisou os conflitos indiretos identificados automaticamente pela ferramenta. O objetivo dessa questão de pesquisa foi descobrir o quanto a estratégia de identificação de conflitos indiretos utilizando grafo de chamadas não apresenta falsos positivos. Foram selecionadas tarefas que apresentavam conflitos indiretos de forma que essas tarefas representassem diferentes cenários de integração.

Após essa seleção, cada tarefa foi analisada manualmente para constatar se realmente o que a ferramenta estava retornando, representavam conflitos indiretos. Para esse estudo foi utilizado um nível de profundidade 3 para o grafo de chamadas. Como pode ser visto a maioria dos conflitos indiretos detectados pela ferramenta encontram-se no nível 1 e 2, o que justifica a análise de conflitos indiretos até o nível 3. Além disso, realizar a análise manual de níveis superiores, tais como 4, 5 ou 6 , é extremamente difícil, pois as vezes é difícil identificar manualmente como artefatos tão distantes, em termos do grafo de chamadas, estão relacionados entre si.

A Tabela 4-10 apresenta a quantidade de conflitos indiretos detectados para a evolução do sistema SIGAA analisada nesse estudo, por níveis de conflitos em que eles foram detectados. No primeiro nível de profundidade, foram detectados 328 conflitos, ou seja, 328 artefatos no *Target* que sofreram alguma mudança referenciavam 328 artefatos a serem integrados no *Source*. No segundo nível de profundidade, 327 artefatos alterados no *Target* referenciavam alguns artefatos intermediários que por sua vez, referenciavam 327 artefatos no *Source* a serem integrados, e assim sucessivamente. Totalizando 847 conflitos indiretos para os três níveis analizados.

Tabela 4-10: Conflitos Indiretos Detectados por Nível de Profundidade da Análise

	Número de Conflitos Indiretos	Porcentagem
Nível 1	328	38,72 %
Nível 2	327	38,61 %
Nível 3	192	22,67 %
Total	847	100,00 %

Para responder essa questão de pesquisa foram analisadas 15 tarefas que envolviam conflitos indiretos. Inicialmente, as 15 tarefas analisadas, 12 indicaram corretamente a presença de conflitos indiretos e 3 apresentaram falsos positivos, indicando artefatos como estando em conflito, quando na verdade não estavam. A correção do problema encontrado nessas últimas 3 tarefas é explicado na Seção 4.4.3.4. As 15 tarefas utilizadas nessa análise estão listadas no Apêndice I deste trabalho.

4.4.3.1 Resultados da análise da terceira questão de pesquisa

O primeiro resultado a se destacar é que todas as 12 tarefas analisadas que possuíam conflitos indiretos, já com a correção discutida na Seção 4.4.3.4, estavam corretas. Os seja, todos os artefatos analisados realmente estavam no grafo de chamada do artefato que se pretende integrar e sofreram evoluções no *Target*. Em outras palavras, foram alterados considerado o código original a partir de onde foram clonados.

Como os artefatos já se apresentam diferentes em relação ao artefatos presentes no *Source*, uma possível integração da mudança realizada no *Source* pode fazer o sistema *Target* passar a se comportar de uma maneira não esperada, assim caracterizando um conflito indireto para a abordagem. Para esses conflitos indiretos detectados realizou-se a mesma verificação feita no caso dos conflitos diretos. Se esses conflitos realmente representavam problemas de integração.

Verificou-se que, para as 12 tarefas indicadas pela ferramenta, 7 tarefas ou 58% apresentavam realmente problemas de integração. A análise das 5 tarefas restantes (42%) mostrou que elas poderiam ser integradas, sem a princípio adicionar erros de execução ao sistema *Target*. Contudo, a constatação de tarefas com problemas de integração não foi contabilizada como um erro e geração de falsos positivos nos resultados do estudo porque envolve análise semânticas das mudanças, que não é o objetivo da análise de conflitos indiretos. O objetivo da análise realizada pela ferramenta é apenas indicar possíveis artefatos no código que devem ser testados após a integração, pois há a possibilidade de ter sido adicionados erros no sistema.

A Tabela 4-11 apresenta o total de tarefas analisadas que possuem conflitos indiretos, assim como quais dessas tarefas apresentam conflitos indiretos que podem afetar ou não o comportamento do sistema quando integradas.

Tabela 4-11: Análise dos conflitos indiretos apresentados pela ferramenta

Total de Tarefas Analisadas com Conflitos Indiretos Indicados pela Ferramenta	Apresentavam Conflitos Indiretos que não afetam o comportamento do sistema	Apresentavam Conflitos Indiretos que afetam o comportamento do sistema
12	5	7

Para a realizar a verificação se o conflito indireto retornado pela ferramenta estava correto, para cada artefato que a ferramenta indicava como estando em conflito, era analisado se existe um caminho entre os artefatos em conflitos e se esses artefatos realmente sofreram alteração no sistema *Source* e no sistema *Target*.

A Figura 4-14 apresenta um exemplo da análise manual do grafo de Chamadas da Tarefa 61549. A ferramenta havia retornado que o método *AlteracaoDadosDiscenteMBean#hasPermissaoAlteracaoCompleta()* do *Source* estava em conflito indireto com o método *MatriculaGraduacaoUFOPAMBean#iniciarSolicitacaoMatricula()* no *Target* no terceiro nível de profundidade. Foi então realiza uma análise manual do código e constatou-se que realmente o método *AlteracaoDadosDiscenteMBean#hasPermissaoAlteracaoCompleta()* era chamado a partir do método *AlteracaoDadosDiscenteMBean#selecionaDiscente()*, que por sua vez era chamado pelo método *AlteracaoDadosDiscenteMBean#iniciarAcessoDiscente()*, até em fim chegamos ao método *MatriculaGraduacaoUFOPAMBean#iniciarSolicitacaoMatricula()*. Da mesma forma a ferramenta indicou que o método *PortalDiscenteAction#execute()* do *Source* estava em conflito indireto com o método *AvaliacaoInstitucionalHelper#aptoPreencherAvaliacaoVigente()* no *Target* com o nível de profundidade 1. A análise manual constatou que o método *PortalDiscenteAction#execute()* chama diretamente o método *AvaliacaoInstitucionalHelper#aptoPreencherAvaliacaoVigente()*.

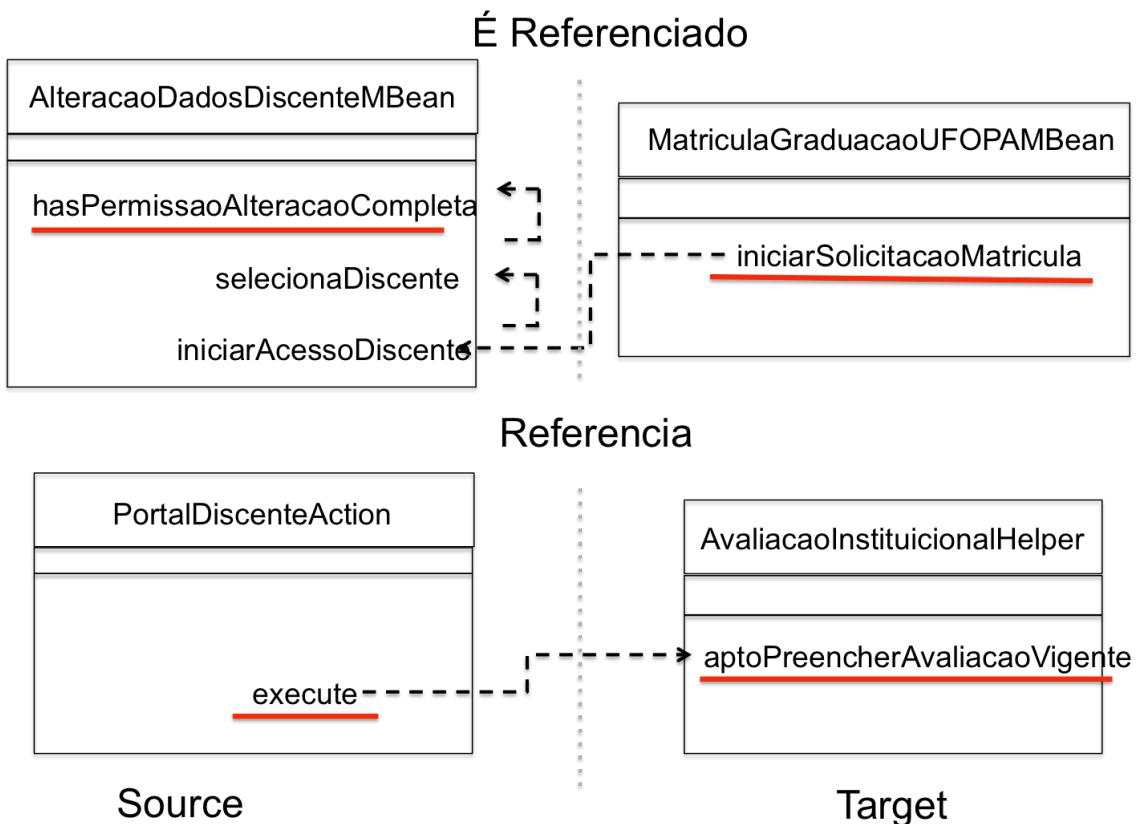


Figura 4-14: Análise do Grafo de Chamadas para a tarefa 61549

Essa análise manual realizada para todos os conflitos analisados demonstrou que o algoritmo desenvolvido para se gerar o grafo de chamadas a partir de referências retornadas pela pesquisa do JDT estava sendo calculado corretamente. Tanto para os artefatos referenciados pelo artefato alterado no *Source*, quanto para os artefatos que referenciam o artefato alterado no *Source*.

A análise do grafo de chamadas para se identificar os conflitos indiretos é realizada na versão final do código do sistema *Target*. Para esse estudo foram analisados 12 tarefas e 40 artefatos indicados pela ferramenta como possuindo conflitos.

Como não foi possível executar e testar massivamente a versão integrada do *Target*, não temos como afirmar que tais conflitos realmente causariam problemas de execução ao sistema *Target*. Percebe-se que com uma análise manual, a identificação de conflitos é uma tarefa bastante árdua reforçando a motivação para criação de abordagens que apoiem tal integração de forma automática. Nas próximas seções são

detalhados exemplos do que são conflitos indiretos que não afetam a semântica final do comportamento do sistema e conflitos indiretos que afetam.

4.4.3.2 Tarefas com conflitos indiretos que não afetam o comportamento do sistema

Para exemplificar as tarefas nas quais foram identificados conflitos indiretos mas que não causariam problemas ao sistema *Target* após a integração, podemos destacar duas tarefas:

Tarefa: 61489 - Registrar data de aula extra na listagem de aulas para cadastro de tópicos

Conforme mostrado na *Figura 4-15*, a alteração ocorrida no *Target* que gerou o conflito indireto indicado pela ferramenta foi apenas uma operação de extração de um trecho de código para um novo método. Não houver qualquer alteração no código extraído. Essa operação considerada uma refatoração [Fowler et al., 2012] e serve apenas para melhorar a qualidade do código fonte, não resultando em nenhuma alteração semântica. Se não há nenhuma alteração semântica, apenas uma nova chamada de método, e considerando também que essa refatoração não quebrou o código do *Source*, não há como o processo de integração em si adicionar erros ao código do *Target* neste caso.

```

* Método chamado pela seguinte JSP: /ava/FrequenciaAluno/formPlanilha.jsp
* @throws ArqException
*/
public void cadastrarPlanilhaAutomatico() throws ArqException {
    try {
        if (turma.isAberta()){
            popularDadosInformados();
        }
        try {
            execute(new MovimentoFrequenciaPlanilha(listagemPlanilha, turma));
            addMensagem(MensagensArquitetura.OPERACAO_SUCESSO);
            calendarios = getCalendarios();
        } catch (NegocioException e) {
            addMensagem(e.getMessage());
        }
    } catch (ArqException e) {
        addMensagem(e.getMessage());
    }
}

/* @throws ArqException */
public void cadastrarPlanilhaAutomatico() throws ArqException {
    try {
        if (turma.isAberta()){
            // Corrigindo problema com CharSet do javascript.
            String [][] valores = converteEmVetor(dadosFrequenciaPlanilha);
            preparaDadosPlanilha();
            String [][] original = converteEmVetor(dadosFrequenciaPlanilha);

            for (int i = 0; i < valores.length; i++){
                if (!valores[i][4].equals("T")){
                    if (!original[i][4].equals(valores[i][4])){
                        original[i][4] = valores[i][4];
                        // Atualiza os valores e indica que foram modificados
                        listagemPlanilha.get(i)[4] = original[i][4];
                    }
                }
            }
        }
    } catch (ArqException e) {
        addMensagem(e.getMessage());
    }
}

```

Figura 4-15: Extração de código fonte para um método

Tarefa 67250 - Erro na criação da turma

Nessa tarefa o método *TurmaMBean#atualizar()* que gerou o conflito indireto foi alterado, porém essa atualização foi apenas uma verificação se alguns dados estavam nulos conforme mostrado na Figura 4-16. Provavelmente para corrigir um erro de *NullPointerException* que estava ocorrendo no método *atualizar()*.

```

if (obj.isGraduacao()){
    if(obj.getTurno() == null)
        obj.setTurno(new Turno());
    if(obj.getCodigoTurma() == null)
        obj.setCodigoTurma(new CodigoTurma());
}

```

Figura 4-16: Alteração do Método *TurmaMbean#atualizar()*

A alteração realizada no *Source* para verificar se a disciplina é um disciplina em bloco, conforme mostra a Figura 4-17, é realizada em um método distinto, pois se trata de um conflito indireto e semanticamente não irá impactar em nada a verificação se os

dados da turma estão nulos ou não, mostrados na Figura 4-16. Ou seja, eu posso aplicar ao *Target* a nova verificação se uma disciplina é em bloco, porque o conflito indireto gerado pela mudança da verificação de dados nulos, não interfere no cálculo das disciplinas em bloco. Portanto, essas alterações podem coexistir.

```
@Override  
public boolean isDefineDocentes() {  
    // se a turma for de módulo, não define docente  
    if (obj.getDisciplina().isBloco())  
        return false;  
    else  
        return true;  
}
```

Figura 4-17: Alteração do método *TurmaGraduacaoMBean#isDefineDocentes()*

4.4.3.3 Tarefas com conflitos indiretos que afetam o comportamento do sistema

Tarefa 51132 - VALIDACAO DE CONSOLIDACAO

Nessa tarefa o método *ProcessadorConsolidacaoTurma#execute()* foi alterado no *Target* para realizar uma validação diferente das regras de negócio para alunos do nível de graduação, conforme pode ser visto na Figura 4-18. O método *ConsolidacaoValidator#validar()* que é chamado por essa classe no *Source* sofreu também uma evolução. Como quem realizou a alteração no *Source* não levou em consideração que existia essa alteração no sistema *Target*, as novas validações podem impactar e fazer com que as mudanças realizadas no *Source* não funcionem do mesmo jeito a serem aplicadas no sistema *Target*.

```

try {
    if (mov.getCodMovimento().equals(SigaalListaComando.CONSOLIDAR_TURMA))
        ConsolidacaoValidator.validar(dao, turma, getParametros(turma),
    }
} finally {
    dao.close();
}

```



```

try {
    if (mov.getCodMovimento().equals(SigaalListaComando.CONSOLIDAR_TURMA))
        if (turma.isGraduacao())
            ConsolidacaoValidatorUFOPA.validar(dao, turma, getParametros(turma),
        else
            ConsolidacaoValidator.validar(dao, turma, getParametros(turma),
    }
}

```

Figura 4-18: Alteração Regras de Validação no Processador

Tarefa 89582 - Gestor Avaliacao nao consegue visualizar o relatorio analítico

Outro exemplo de uma tarefa que apresentou um conflito semântico com uma análise manual foi a tarefa de número 89582. A Figura 4-19 mostra que no sistema *Source*, a busca do método *AvaliacaoInstitucionalDao#findEvolucaoMediaGeralAnoPeriodo(int, boolean)* foi alterada para ser adicionado mais um critério na busca realizada.

```

    * @throws DAOException
 */
public Map<String, Double> findEvolucaoMediaGeralAnoPeriodo(int idServidor, boolean somenteResultadosLiberados)
{
    String sql = "select resultado_avaliacao_docente.ano, resultado_avaliacao_docente.periodo, " +
        " avg(media_notas.media) as media_geral" +
        " from avaliacao.resultado_avaliacao_docente" +
        " inner join avaliacao.media_notas using (id_resultado_avaliacao_docente)" +
        " inner join avaliacao.pergunta on (pergunta.id = media_notas.id_pergunta)" +
        " left join avaliacao.parametro_processamento_avaliacao using (ano, periodo)" +
        " inner join ensino.docente_turma using (id_docente_turma)" +
        " inner join ensino.turma t using (id_turma)" +
        " left join ensino.docente_externo on (docente_externo.id_docente_externo = docente_turma.id_docente_externo)" +
        " left join rh.servidor on (servidor.id_servidor = docente_turma.id_docente or servidor.id_servidor =
        " where (t.distancia is null or t.distancia = falseValue() or t.id_polo is null)" +
        " and servidor.id_servidor = :idServidor" +
        " and pergunta.id_grupo = :idGrupoPergunta" +
        "(somenteResultadosLiberados ? " and consulta_docente = trueValue() : "")" +
        " group by resultado_avaliacao_docente.ano, resultado_avaliacao_docente.periodo" +
        " order by resultado_avaliacao_docente.ano, resultado_avaliacao_docente.periodo";
}

```

Figura 4-19: Adição de um campo na busca no sistema Source

Os métodos *PortalResultadoAvaliacaoMBean#paginaInicial()* e *PortalResultadoAvaliacaoMBean#carregarMedias()* que chamam indiretamente o método *AvaliacaoInstitucionalDao#findEvolucaoMediaGeralAnoPeriodo(int, boolean)* a partir do nível 3 e 2 de profundidade respectivamente sofreram evoluções no *Target*. A Figura 4-20 mostra evolução ocorrida do método *PortalResultadoAvaliacaoMBean#carregarMedias()*. O método original do *Source* considerava que os resultados retornados seriam filtrados por determinados critérios, as alterações realizadas no *Target* podem passar a darem problema, se os resultados retornados não forem os esperados quando elas foram implementadas. Por exemplo, os métodos do *Target* poderiam ter sido alterados para mostrar para o usuário determinadas informações da lista de resultados retornados. Com a aplicação da restrição da consulta do *Source*, essas informações não poderiam ser retornadas, gerando uma exceção ao tentar acessá-las.

```

<4b>
247  /**
248  ** Customização UFOPA - SIGProject 22387 */
249  private void carregaMedias() throws HibernateException, DAOException
250  {
251     if( dto == null )
252         mediaGeralSemestre = 0;
253     else
254         dto.setMediaGeralSemestre(0);
255
256     AvaliacaoInstitucionalDao dao = getDAO(AvaliacaoInstitucionalDao.
257     List<ResultadoAvaliacaoDocente> resultadosDocentes = dao
258             .findResultadoByDocenteCentroDepartamentoAnoPeriodo(servi
259
260     if( dto == null )  this.resultadosDocentes = resultadosDocentes;
261     else
262         dto.setResultadosDocentes(resultadosDocentes)
263
264     if (resultadosDocentes == null || resultadosDocentes.isEmpty())
265     {
266         addMensagemErro("Não há registro de médias de avaliações para
267         return;
268     }
269     String anoPeriodo = parametroProcessamento.getAno()+"."+parametro

```

Figura 4-20: Customização do sistema *Target*, que é afetada indiretamente pela mudança do *Source*.

Como a quantidade de alterações sofridas pelo método da Figura 4-20 é significativa, mesmo com a análise manual por um ser humano, é difícil ter certeza se a mudança causará um comportamento inesperado no sistema *Target*. Neste caso, somente realizando a integração e se executando vários testes para essa funcionalidade teríamos uma maior probabilidade de que, pelo menos para os comportamento testados,

há ausência de erros quando considerando os mesmos testes executados para a versão anterior do sistema *Target*.

4.4.3.4 Solução para análise de artefatos genéricos

A análise realizada no nosso estudo também mostrou em alguns casos que, mesmo não existindo erro no algoritmo que gera o grafo de chamadas e identifica os artefatos relacionados com artefato alterado, eram detectados falsos positivos. Isso ocorreu nas tarefas de número 72552, 80445 e 71885 (detalhes no Apêndice I). Em outras palavras, a ferramenta retornava determinados artefatos como estando em conflito indireto, quando não possuíam nenhuma relação sintática ou semântica.

Isso ocorria porque para algumas superclasses, que possuem métodos abstratos que eram sobreescritos e utilizados por várias classes do sistema, o JDT retorna uma quantidade muito grande de artefatos relacionados. A Figura 4-21 mostra um exemplo desse comportamento do JDT.

The screenshot shows the Eclipse IDE interface with the Java Search view open. The title bar reads "'br.ufrn.sigaa.pesquisa.dominio.GrupoPesquisa.toString()' - 797 references in workspace (no JRE) (0 matches filtered from view)'. The search results list 797 references, each showing a file path and a line number. The results are as follows:

- br.edu.ufopa.sigaa.selecaoexterna.jsf - selecao_interna - SIGAA_TARGET 79062
- br.ufrn.arq.dao - src - 01_Arquitetura 178855
- br.ufrn.arq.email - src - 01_Arquitetura 174785
- br.ufrn.arq.migracao - src - 01_Arquitetura 177565
- br.ufrn.arq.util - arq - SIGAA 177120
- br.ufrn.arq.util - src - 01_Arquitetura 178586
- br.ufrn.arq.web.jsf - src - 01_Arquitetura 176520
- br.ufrn.arq.web.jsf - web - 02_EntidadesComuns 165543
- br.ufrn.arq.web.struts - src - 01_Arquitetura 149500
- br.ufrn.arq.web.tags - src - 01_Arquitetura 176876
- br.ufrn.banco.reversa.utils - services - 01_Arquitetura 178087
- br.ufrn.comum.jsf - web - 02_EntidadesComuns 166266
- br.ufrn.integracao.seguranca - services - 01_Arquitetura 176876
- br.ufrn.integracao.servicos - integracao - 03_ServicosIntegrados 162935
- br.ufrn.integracao.signed.dto - integracao - 03_ServicosIntegrados 164657
- br.ufrn.sigaa.apedagogica.jsf - apedagogica - SIGAA 176885
- br.ufrn.sigaa.apedagogica.jsf - apedagogica - SIGAA_TARGET 26753
- br.ufrn.sigaa.arq.ajax - geral - SIGAA 177152
- br.ufrn.sigaa.arq.ajax - geral - SIGAA_TARGET 79062
- br.ufrn.sigaa.arq.dao - arq_dao - SIGAA 179062
- br.ufrn.sigaa.arq.dao - arq_dao - SIGAA_TARGET 79062
- br.ufrn.sigaa.arq.dao.ead - arq_dao - SIGAA 170923
- br.ufrn.sigaa.arq.dao.ensino - arq_dao - SIGAA 178957

Figura 4-21: Busca JDT para o método abstrato *toString()*

Todo objeto em Java possui um método *toString()* que normalmente é utilizado para imprimir informações sobre o objeto, em outras palavras, transformar os dados do

objeto em uma *string* para que possa ser impresso na tela ou no console. Percebe-se que ao se tentar recuperar os artefatos que chamam o método *toString()* de um objeto do sistema utilizado no estudo, 797 referências são retornadas. Isso ocorre porque a maioria dos objetos do domínio da aplicação sobrescrevem esse método. O JDT busca todas as referências para o método abstrato da classe pai *Object#toString()*. Então, praticamente toda classe do sistema que faz uso de algum objeto do domínio, que normalmente sobrescreve esse método, é retornada como possuindo uma referência ao método do objeto selecionado.

Qualquer alteração realizada nesse método, retorna uma quantidade muito grande de artefatos relacionados. Isso é um comportamento do JDT que não podemos alterar e que acarreta muitos falsos positivos para a análise de conflitos indiretos. Para solucionar esse problema foi escolhida uma estratégia já presente em outras ferramentas de análise estática de código, como o WALA [IBM, 2015]. Foi criado um arquivo onde pode-se adicionar artefatos que devem ser ignorados durante a análise de conflitos indiretos. A *Figura 4-22* mostra a lista de artefatos, separados por ponto e vírgula, que foram ignorados durante a análise presente neste trabalho.

```
#getId();#setId(int);#toString();$int:id;#equals(Object)
```

Figura 4-22: Artefatos ignorados durante a análise desse trabalho

Como pode ser visualizado na *Figura 4-22*, os métodos genéricos presentes em todos os objetos Java *toString()* e *equals(Object)*, além dos métodos genéricos *getId()*, *setId()* e o campo *id* presentes em todos os objetos do domínio da aplicação, foram ignorados na análise. No algoritmo de construção do grafo de chamadas da análise de conflitos indiretos, foi adicionada a validação presente na Listagem 4.2.

Listagem 4.2. Incremento do Algoritmo de Detecção de Conflitos Indiretos

Para cada artefato presente no ChangeLogHistory da LPS Source, que **não** está em conflito direto

 Se o artefato **não** está presente na listagem de artefatos ignorados

 Calcula o grafo de chamadas desse artefato no Sistema *Target* ...

/ demais passos do algoritmo continuam os mesmos e foram omitidos */*

 Se não

 Passa para o próximo artefato

Com essa mudança no algoritmo e a utilização da lista de artefatos que deveriam ser ignorados durante a análise do grafo de chamadas, mostrados na *Figura 4-22*, a quantidade de conflitos indiretos detectados pela ferramenta diminuiu em 62%, passando de 2223 para 847 conflitos indiretos.

Após essa diminuição, as tarefas 72552, 80445 e 71885, da Tabela I-3, que haviam sido reportadas como possuindo conflitos indiretos, e pela análise manual constatou-se que não existiam esses conflitos, passaram a serem reportadas corretamente, como não possuindo conflitos. Além disso foi notado que algumas tarefas, como por exemplo as tarefas 80445 e 71885 utilizadas nesse estudo, apresentaram um número muito grande de conflitos indiretos, destoando das demais, consequência da característica da pesquisa JDT. Esse problema também foi corrigido.

A inclusão de artefatos que devem ser ignorados durante a análise pode ter omitidos alguns conflitos, gerando resultados falsos negativos para a análise. Pela quantidade muito grande de tarefas sem conflitos que precisariam ser analisadas para se verificar se existiriam resultados falsos negativos, essa análise não foi possível de ser realizada nesse trabalho.

4.5. Limitações e Ameaças ao Estudo

4.5.1 Ameaças Internas:

As verificações de integridade do sistema após a realização do *merge* foram limitadas à verificação de erros de compilação e uma análise do código para determinar que a aplicação das alterações selecionadas não causariam erros no sistema. Infelizmente, não foi possível verificar o funcionamento correto do sistema *Target* após a integração, pela não disponibilidade da versão do banco de dados após o *merge* que

permitisse executar o sistema, como também não foi possível nem executar testes automatizados no sistema, pois os sistemas analisados até as versões utilizadas no estudo não dispunham de tais testes⁴.

Na segunda e na terceira questão de pesquisa foi realizada uma análise de código para determinar quais tarefas poderiam ser integradas mesmo possuindo conflitos diretos e quais conflitos indiretos afetariam o comportamento do sistema após a integração. Essa análise foi realizada apenas pelo autor desta dissertação e está, portanto, sujeita a erros. Para minimizar tal ameaça, tal análise foi realizada de forma meticulosa e sem limite de tempo, garantindo um bom intervalo para análise de cada tarefa, além de também ter sido realizada pelo autor que é um desenvolvedor com bastante experiência e que faz parte da equipe do SIGAA.

Na seleção das tarefas para o estudo não foi realizado nenhum sorteio que eliminasse alguma tendência na seleção das tarefas. Elas foram selecionadas subjetivamente obedecendo apenas os critérios de agrupamento explicados na Seção 4.3 para tentar representar diferentes cenários de integração.

4.5.2 Ameaças Externas:

Os resultados do estudo também estão restritos ao sistema e evolução analisados, não podendo os mesmos serem generalizados para diferentes cenários de evolução ou sistemas que possuam características diferentes do analisado. Havendo, portanto, a necessidade de replicar o estudo para outros contextos de forma a expandir a generalidade de suas conclusões. O código fonte da ferramenta encontra-se disponível no endereço <https://github.com/jadsonjs/MergeClear> para quem desejar utilizá-la em outros estudos.

⁴ Atualmente, a SINFO/UFRN vem investindo no desenvolvimento de testes automatizados tanto no nível de unidade/integração com o framework JUnit, quanto no nível de sistemas com a ferramenta Selenium

5. Trabalhos Relacionados

Trabalhos de pesquisas recentes demostram que a abordagem *clone-and-own* ainda é uma alternativa viável em alguns cenários. Essa abordagem gera, entre outros problemas, conflitos de integração de código fonte. Pesquisas recentes propõem algoritmos para identificar tais conflitos e estratégias para sua resolução. Porém esse problema ainda se mostra desafiador, estando longe de uma solução viável. As subseções seguintes apresentam trabalhos relacionados à abordagem *clone-and-own* e à classificação e resolução de conflitos.

5.1. Trabalhos relacionados à Resolução de Conflitos

Guimarães & Silva [Guimarães & Silva. 2012] desenvolveram uma abordagem de *merge* contínuo que analisa os conflitos que vão sendo gerados a medida em que o desenvolvedor altera o código fonte do sistema no seu *workspace* local. O objetivo dessa análise é corrigir, o mais rapidamente possível, os conflitos gerados antes que o desenvolvedor tente enviar as suas mudanças para o repositório de código, onde os demais desenvolvedores também estão realizando alterações.

Essa abordagem se aplica a um contexto diferente do nosso, mas várias das análises e classificações de conflitos realizadas podem ser aplicadas ao nosso problema. A análise de conflitos se dá a partir da construção e comparação de uma árvore de elementos da linguagem (pacotes, classes, atributos e métodos), mais simplificada do que a árvore de sintaxe abstrata (AST- Abstract Syntax Tree), do *workspace* local do desenvolvedor com a versão remota (publicada no repositório) da árvore. Essa comparação detecta alguns conflitos, tais como:

- **Conflitos estruturais:** conflitos encontrados durante o *merge* executado em segundo plano a partir da análise das árvores de elementos da linguagem. Estes sub conflitos são, por sua vez, subdivididos em:
 - Pseudo conflitos diretos: ocorre quando diferentes elementos de um mesmo nó da árvore são modificados ou quando são alterados para um mesmo valor.
 - Conflito de mudança de atributos e métodos: ocorre quando o mesmo atributo/método é alterado.

- Conflito de mudança e remoção de nó: ocorre quando um desenvolvedor altera o método e outro o deleta.
- Conflito de tipo de nó inconsistente: ocorre na tentativa de adicionar nós de diferentes tipos em uma mesma localização.
- **Conflitos de Linguagem:** após o *merge*, se existirem erros de compilação, esses erros são denominados de conflitos de linguagem.
- **Conflitos comportamentais:** são os conflitos que ocorrem no grafo de chamadas do elemento que está sendo alterado. A abordagem analisa as dependências do nó que foi modificado e verifica se no caminho não há mudanças que conflitem com a que foi realizada no nó.
- **Conflitos de teste:** representa uma falha de teste ocorrida após a execução do *merge* e da execução dos testes unitários automatizados.

Os algoritmos de identificação de conflitos nessa abordagem poderiam ser integrados a ferramenta para serem utilizados no problema de *merge* tratado por este trabalho e comparado aos resultados obtidos.

Brun e outros [Brun et al. 2013] desenvolveram uma técnica de análise especulativa que analisa as mudanças que estão ocorrendo no código local do desenvolvedor, para tentar prevê possíveis conflitos antes deles serem compartilhados com outros desenvolvedores no controle de versão. Eles classificam os conflitos em dois tipos de conflitos: (i) os conflitos textuais - que representam alterações inconsistentes que dois desenvolvedores realizam para a mesma parte do código-fonte e (ii) os conflitos de ordem superior - quando não há conflitos textuais entre as alterações dos desenvolvedores, mas essas mudanças são incompatíveis semanticamente. Esses conflitos de ordem superior causam erros de compilação, falhas nos testes, ou outros problemas, e são difíceis para detectar e resolver na prática.

A análise especulativa antecipa as ações que um desenvolvedor pode querer executar e as executa em segundo plano. Quando aplicado a um desenvolvimento colaborativo com o uso de controle de versão, a análise especulativa pode usar informações anteriormente não exploradas para diagnosticar precisamente importantes classes de conflitos e oferecer sugestões concretas sobre como lidar com eles. Relatando as consequências destas operações similares a como um sistema controle de versão reportaria, com a finalidade de melhorar a maneira em que os desenvolvedores

colaborativos identificam e gerenciam os conflitos. Estas informações também permitem que os desenvolvedores tomem melhores decisões sobre como e quando compartilhar mudanças, enquanto simultaneamente reduz a necessidade de raciocínio e processamento humano. Para o estudo conduzido [Brun et al. 2013] foram analisados nove sistemas de código aberto nos quais foi constatado que conflitos entre cópias dos desenvolvedores de um projeto são normais, e não a exceção. Além disso, foi também identificado que 33% dos conflitos são de ordem superior.

Tal trabalho se destina a resolução de conflitos em ambiente colaborativo de desenvolvimento não focando no problema de *merge* entre sistemas clonados. Os autores apresentam uma avaliação quantitativa e preliminar da abordagem, diferentemente deste trabalho que avaliou qualitativamente os resultados obtidos pela abordagem proposta por [Lima, et al. 2013]. Os conflitos identificados por essa abordagem também poderiam ser adaptados para serem integrados ao *MergeClear* e comparados aos resultado obtidos neste estudo.

Apel e outros [Apel, S., Lieig, J., Kästner, 2011] propuseram um sistema de controle de revisão semiestruturado que herdam as características fortes dos sistemas de revisão não estruturados, como a independência da linguagem, e dos sistemas de revisão estruturados, como a expressividade. Porém sem suas fraquezas. O trabalho utiliza anotações de código para fornecer informações estruturais dos artefatos de software. Desta forma, uma ampla variedade de linguagens podem ser abordadas e as informações fornecidas por essas anotações aumentaram a quantidade de informação que um sistema de controle de revisão tem à sua disposição para resolver conflitos.

Os autores desenvolveram uma *engine* genérica, chamada FSTMerge, que realiza o *merge* de diferentes revisões de um sistema de software com base na estrutura dos artefatos de software envolvidos. Os usuários podem conectar uma nova linguagem ao FSTMerge, fornecendo uma especificação declarativa de sintaxe da sua linguagem. Se não existe uma gramática especificada, o FSTMerge pode usar uma solução que analise o código linha por linha, equivalendo a um *merge* não estruturado.

Os autores conduziram um estudo em 24 sistemas escritos em Java, C# e Python abrangendo 180 cenários de *merge*. Eles verificaram que um *merge* semiestruturado reduziu o número de conflitos no geral em 60% em comparação com os sistemas não

estruturados. Eles também encontraram que a operação de renomear é um problema para o *merge* semiestruturado e a combinação de *merge* semiestruturado com o não estruturado é o melhor caminho para balancear precisão com desempenho, reduzindo o número de conflitos em 35% em relação ao uso puro *merge* não estruturados.

Esse trabalho foi utilizado como base para decidir pelo uso de um *merge* estruturado na abordagem de [Lima, et al. 2013]. Como, devido a complexidade, a reconciliação entre sistemas clonados costuma ocorrer entre períodos grandes de evolução, a precisão dos resultados era mais importante do que o tempo para se analisar os dados gerados pelo *merge* estruturado, já que a reconciliação de sistema clonados é uma tarefa complexa que não costuma ser realizada constantemente.

5.2. Trabalhos relacionados à Clonagem de Sistemas

Rubin et al [Rubin et al, 2012] propõem uma abordagem para utilizar as vantagens existentes no uso da técnica de clonagem mitigando as desvantagens do uso desse mecanismo. Para isso, eles definem um modelo chamado de Product Line Changeset Dependency Model (PL-CDM). PL-CDM armazena grupos de modificações relacionadas do código fonte em funcionalidades e esses grupos de funcionalidades em grupos de *features*, além de relações de dependência entre essas funcionalidades e as *features* presentes no PL-CDM.

PL-CDM é construído por meio da extração e análise dos dados contidos no repositório SCM e, possivelmente, outras fontes de informação, tais como um sistema de Gestão da Mudança (CM). Esta análise de dados tem os seguintes objetivos:

1. *Agregação de Features*: agrupamento de alterações de código incrementais em conceitos de nível superior de SPLE, como *features*. Isso facilita a elevar o nível de abstração e raciocínio sobre a linha de produtos desenvolvidos usando a aceitável terminologia SPLE e mecanismos de gestão.
2. *Cálculo de Dependência de Features*: a detecção de dependências semânticas entre implementações de *features*, tais como as dependências require indicando que uma *feature* não pode ser realizada sem a outra. Isto facilita o isolamento do essencial funcionalidade mínima para uma *feature* possa ser operada.

Nesse trabalho os autores apresentam uma abordagem genérica e preliminar para

resolução de *merge* em LPSSs clonadas. Nenhum avaliação é realizada a respeito da abordagem proposta. Em um trabalho posterior [Rubin, Czarnecki and Chechik 2013] os autores realizaram um estudo empírico envolvendo três organizações que utilizam LPSSs clonadas e analisaram em detalhe as atividades de desenvolvimento realizadas por essas empresas. Diferentemente deste trabalho, nenhuma avaliação é apresentada sobre a abordagem proposta para resolução de *merge*.

Dubinsky e outros [Dubinsky et al. 2013] realizaram um estudo exploratório para avaliar a cultura de clonagem no contexto de linhas de produtos de software. Especificamente realizaram onze entrevistas com os participantes envolvidos em seis linhas de produtos clonados de três grandes organizações bem estabelecidas, que fornecem soluções em armazenamento de dados, aeroespacial e de defesa, e de domínio automotivo. Foram entrevistados funcionários de diversas funções, como desenvolvedores, testadores e gerentes de produto. Foi utilizado um questionário estruturado para complementar os dados da entrevista.

Entre os resultados, esse estudo observou que a maioria das linhas de produtos que usam técnicas de clonagem ainda enxerga como uma abordagem de reutilização favorável, principalmente porque é um mecanismo rápido que permite que os profissionais começar a partir de uma funcionalidade já especificadas e verificadas, mantendo a liberdade e independência para fazer as alterações necessárias. De fato foi constatado que vários praticantes estão satisfeitos com a prática de clonagem e acreditam que é um mecanismo de reutilização viável. Enquanto outros gostariam de mudar para uma abordagem mais bem gerenciada. Observou-se que as estruturas organizacionais existentes podem impedir essa mudança, porque normalmente não há nenhuma regra mas organizações avaliadas que seja responsável por promover a reutilização das funcionalidades de software. O estudo realizado não faz menção a nenhuma ferramenta para resolução de *merge* entre LPSSs clonadas. Nem tão pouco realiza uma avaliação como a avaliação qualitativa conduzida nesta dissertação.

Lima [Lima, 2014] realizou um estudo quantitativo utilizando a abordagem presente nesta dissertação. O foco do trabalho foi levantar dados sobre os conflitos que ocorreram entre quatro evoluções de duas LPSSs nas quais era aplicada a abordagem de *clone-and-own*. Foram levantadas várias informações de como os conflitos detectados pela abordagem ocorreram nas evoluções estudadas, por exemplos: (i) os tipos de

tarefas que apresentaram mais conflitos, os tipos e quantidade de conflitos que ocorreram nas camadas da arquitetura das LPSs estudadas, entre outras informações. Este trabalho focou em analisar a qualidade dos resultados retornados pela ferramenta. Foram realizadas algumas melhorias no grafo de chamadas gerado pela ferramenta, essas melhorias permitiram a geração do grafo completo de chamadas bem como eliminou falsos positivos que a análise anterior apresentava.

O estudo apresentado neste trabalho também mostrou que, para o cenário avaliado, a identificação de conflitos com base do grafo de chamadas pode auxiliar, na maioria dos casos, o desenvolvedor a ter uma maior certeza se a integração da tarefa causará algum problema de compilação ao sistema *Target*, como também revela possíveis trechos de código, identificados pelos conflitos indiretos, que podem apresentar problemas na integração e precisam ser testados com maior cuidado.

6. Considerações Finais e Trabalhos Futuros

Este trabalho apresentou um estudo quantitativo que buscou avaliar a abordagem proposta por [Lima, et al. 2013] para reconciliação de sistemas clonados baseadas em análise de conflitos. O objetivo principal desse trabalho foi avaliar se a mineração e análise de conflitos realizados pela ferramenta estavam obtendo informações corretas, além de verificar se a identificação dos tipos de conflitos definidos era condição suficiente para o *merge* ocorrer de uma das três formas que foram proposta na abordagem: (i) automática: para tarefas sem conflitos, (ii) semiautomática: para tarefas que apresentavam apenas conflitos indiretos; e (iii) manual: para tarefas com conflitos diretos. Como objetivo secundário, foi avaliado que tipos de conflitos ocorrem e a complexidade desses conflitos ao se tentar realizar a integração de clones de um sistema web de larga escala.

Para isso foram selecionados dados de evoluções do sistema SIGAA realizadas de forma paralela para adicionar variabilidades não presentes na versão *Source* do sistema a partir da qual a versão *Target* foi clonada. Para o *Source* foram analisadas pela ferramenta as evoluções ocorridas entre as versões 3.6.0 e 3.8.0, totalizando 1083 tarefas. Já no lado do sistema *Target*, foram analisadas 99 tarefas realizadas entre as versões 3.6.0 e 3.7.64. A versão 3.6.0 foi a primeira versão do *Target* na qual foram registradas evoluções, assim tem-se a garantia que todas as mudanças realizadas no código foram analisadas pela ferramenta.

Foram determinadas três questões de pesquisa e para cada questão foi selecionado um conjunto de tarefas para serem analisadas de forma manual, tentando simular uma integração manual dessas tarefas, e comparando com os resultados reportados pela ferramenta. Apesar da limitação da pouca quantidade de tarefas que foram analisadas, a ferramenta apresentou bons resultados os quais são discutidos a seguir.

6.1. Análise das Questões de Pesquisa da Dissertação

Para responder a primeira questão de pesquisa – “*Tarefas identificadas pela abordagem como não tendo conflitos podem de fato ser integradas?*” – foi idealizada uma análise qualitativa de um conjunto de tarefas indicadas pela ferramenta como não

possuindo nenhum dos tipos de conflitos definidos na abordagem avaliada. Os resultados obtidos mostraram que, de uma forma geral, a grande maioria das tarefas podem ser integradas. As que não puderam, apresentaram erros no cálculo das dependência entre tarefas ou as informações de evolução foram geradas de forma errada durante o processo de desenvolvimento, o que gerou ruídos para a análise da ferramenta avaliada neste trabalho.

Para responder a segunda questão de pesquisa – “*Tarefas com conflitos diretos não podem realmente ser integradas ? Qual a complexidade da integração de tarefas indicadas como possuindo conflitos diretos pela abordagem avaliada?*” – foi realizada outra análise quantitativa com tarefas que indicavam conflito diretos. Para essa questão de pesquisa a ferramenta demonstrou que está recuperando os resultados corretamente para todas as tarefas. Uma análise manual de código realizada para o estudo, demonstrou que alguns desses resultados, apesar de serem conflitos diretos são de fácil resolução, enquanto outros são complexos de serem resolvidos até manualmente.

Para responder a terceira e última questão de pesquisa – “*Os conflitos indiretos identificados pela abordagem representam conflitos que trazem problemas para a integração?*” – foi realizada mais uma análise manual de um conjunto de tarefas identificadas pela ferramenta como possuindo conflitos indiretos. Os resultados da análise manual demonstraram que a ferramenta também teve um alto grau de precisão para as tarefas selecionadas no estudo, indicando os conflitos indiretos corretamente. Da mesma forma que a questão de pesquisa anterior, a análise manual demonstrou que parte dos conflitos indiretos identificados não são conflitos de fato, pois não adicionam erros no sistema *Target*, enquanto outros muito provavelmente gerariam erros no sistema *Target* após a reconciliação.

6.2. Revisão das Contribuições

Esta dissertação de mestrado possui as seguintes contribuições principais:

- **Avaliação da qualidade dos resultados da abordagem proposta:** A contribuição mais importante para o aspecto científico do trabalho foi a análise da qualidade dos resultados apresentados pela ferramenta. Eles se mostraram satisfatórios em relação à análise manual realizada, constatando que a análise de conflito com base em grafo de chamadas entre artefatos de código, apesar de

não envolver análises semânticas, pode ser uma alternativa viável para encontrar problemas de integração em vários casos.

- **Melhoramento do algoritmo utilizado para a detecção dos conflitos indiretos:** Outro ponto que a avaliação quantitativa possibilitou foi o melhoramento do algoritmo utilizado para a detecção dos conflitos indiretos. Eliminando uma quantidade significativa de falso positivos que a ferramenta apresentava nas versões anteriores.
- **Entendimento dos tipos e complexidade dos conflitos gerados:** A análise manual possibilitou uma visão dos conflitos que ocorreram ao se tentar integrar dois clones de um sistema web de larga escala. Com pôde ser visto, alguns conflitos gerados são bastante complexos, que mesmo uma análise manual realizada por um desenvolvedor não garante que eles possam ser resolvidos.
- **Integração com a ferramenta ChangeDistiller [Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. 2007].** Essa integração pode delegar ao *ChangeDistiller* a identificação das evoluções ocorridas nos elementos de uma classe (métodos, campos, herança) a partir de duas versões do código fonte dessa classe. Essa integração possibilita o uso de uma ferramenta mais madura para identificação de evoluções do código fonte dos sistemas.
- **Adição da visualização da árvore de conflitos:** Essa melhoria possibilitou que os usuários finais da ferramenta visualizarem os resultados completos da análise de conflitos realizada pela ferramenta. Além de quais artefatos de código possuem conflitos já mostrados na visualização principal da ferramenta, os usuários também podem visualizar quais artefatos no *Target* os artefatos do *Source* estão apresentando conflitos.
- **Adição do cálculo de dependências entre tarefas:** O estudo conduzido nesta dissertação também possibilitou demonstrar que, para a integração correta das evoluções em LPS clonadas, tão importante quanto a determinação de conflitos de integração entre as tarefas é determinar aa ordem em que essas tarefas podem ser aplicadas considerando suas dependências.
- **Adição do suporte ao controle de versão GIT:** A ferramenta foi aprimorada para permitir realizar minerações em sistemas que tenham seu código fonte armazenados no sistema de controle de versão distribuído GIT, o qual, vem

ganhando muita popularidade ultimamente, com isso amplia-se a gama de sistemas que a abordagem pode ser aplicada.

- **Adição do suporte aos sistemas de gerenciamento de configuração e de mudanças GitHub e Redmine:** Também com o objetivo que ampliar a quantidade de sistemas onde se poderia aplicar a abordagem proposta, foi adicionado suporte a dois sistemas de gerenciamento de configuração e mudanças *open Source* utilizando por milhares de sistemas e projetos de escala mundial.

6.3. Limitações do Trabalho

O trabalho apresentado nesta dissertação possui várias limitações, dentre as quais podemos destacar:

- As verificações de integridade do sistema após a realização do merge foram limitadas à verificação de erros de compilação e análise do código para determinar que a aplicação das alterações selecionadas não causariam erros ao sistema *Target*.
- A quantidade relativamente pequena de tarefas analisadas que entraram na versão final do estudo.
- Os resultados estão restritos ao sistema e evolução analisada, não podendo ser generalizados para diferentes cenários de evolução ou sistemas.
- As análises manuais realizadas para a segunda e terceira questões de pesquisa, para se determinar quais tarefas poderiam ser integradas mesmo possuindo conflitos diretos e quais conflitos indiretos afetariam ou não o comportamento do sistema foram realizadas pelo autor desta dissertação de forma subjetiva, não existiu um critério formal de identificação de um erro, portanto, essa análise está sujeita a erros.
- A seleção das tarefas analisadas no estudo não seguiu um critério impessoal de seleção, como um sorteio.

6.4. Trabalhos Futuros

Existem vários trabalhos futuros que podem ser desenvolvidos como consequência das pesquisas conduzidas nesta dissertação de mestrado. Abaixo listamos uma série de trabalhos que podem ser conduzidos nessa direção:

- **Implementar as demais fases da abordagem:** Quando a abordagem foi concebida contava com quatro fases, além da mineração e análise de conflitos, existem ainda: (i) a fase de realização do *merge*, que é a aplicação dos trechos de código fonte evoluídos no *Source* no código do sistema *Target* de forma automática, e (ii) a fase indicação de possíveis testes automatizados a serem executados para os artefatos que foram identificados como possuindo conflitos indiretos. Até o momento apenas as duas primeiras fases da abordagem foram implementadas.
- **Replicar esse estudo em diferentes tipos de sistemas e diferentes contextos de evolução:** Para se ter mais confiança e poder de generalização dos resultados encontrados nesse estudo seria interessante replicá-lo em diferentes sistemas. Até mesmo para aumentar o interesse pela ferramenta, é importante que a ferramenta seja aplicada e avaliada a outros sistemas, que não os de desenvolvimento pela Superintendência de Informática da UFRN.
- **Melhorar o algoritmo de detecção de dependência entre tarefas:** Como foi demonstrada no estudo, a identificação correta da ordem em que as evoluções foram realizadas é uma tarefa fundamental para a integração correta das funcionalidades. Essa identificação de dependências entre tarefas não pode ser considerada uma tarefa trivial e ela por si só pode gerar vários novos estudos interessantes.
- **Melhorar a usabilidade da ferramenta:** Para uma aplicação industrial da ferramenta ainda é preciso também melhorá-la visualmente, de forma que, os esforços para configurar e executar as análises de evolução de sistemas clonados seja mínima.

Referências Bibliográficas

Alencar, D. Avaliação da Contribuição de Desenvolvedores para Projetos de Software Usando Mineração de Repositórios de Software e Mineração de Processos. Natal, RN: 2013. 156 f. Dissertação (Mestrado) - Universidade Federal do Rio Grande do Norte.

Antkiewicz, M. et al. Flexible Product Line Engineering with a Virtual Platform. Icse Companion'14. Hyderabad. 2014.

Apel, S., Lieig, J., Kästner, C. 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 (ESEC/FSE '11).

Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. IEEE Transaction on Software Engineering, 33(11), pp. 725-743, 2007

Berczuk, S., Appleton, B. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. 2002.

Brun Y., et al. Early Detection of Collaboration Conflicts and Risks, IEEE Trans. on Software Engineering, vol. 39, no. 10, pp. 1358-1375.

Clements, P., and L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley Professional, 2001.

Czarnecki, Krysztof; Eisenecker, Ulrich. Generative Programming: Methods, Tools, and Applications. New York: Addison-wesley Professional, 2000. 864 p.

DIRECTED acyclic graph. , 2015. Disponível em: <[http://https://en.wikipedia.org/wiki/Directed_acyclic_graph](https://en.wikipedia.org/wiki/Directed_acyclic_graph)>. Acesso em: 18 jun. 2015.

Dubinsky Y., et al. An Exploratory Study of Cloning in Industrial Software Product Lines. 17th European Conference on Software Maintenance and Reengineering (CSMR), 2013

Ferreira, F., et al. Making Software Product Line Evolution Safer. 6th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2013), Natal, Brazil, IEEE Computer. 2012

Fowler, M. Continuous Integration. 2006. Disponível em: <http://martinfowler.com/articles/continuousIntegration.html>. Acessado em 14 de jan de 2014.

Fowler, M. et al. Refactoring: Improving the Design of Existing Code. Addison-wesley Professional, 2012. 464 p.

Guimarães, M. L. & Silva, A. R, 2012. Improving early detection of software merge conflicts. InProceedings of the 2012 International Conference on Software Engineering (ICSE 2012). IEEE Press, Piscataway, NJ, USA, 342-352

IBM T.J. WATSON RESEARCH CENTER. WALA: T.J. Watson Libraries for Analysis. 2015. Disponível em: <http://wala.sourceforge.net/wiki/index.php/Main_Page>. Acesso em: 15 jul. 2015.

KANG, K. C.; KIM, S.; LEE, J.; KIM, K.; SHIN, E.; HUH, M. Form: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering, v. 5, p. 143–168, 1998.

Krueger, C., Easing the transition to software mass customization. PFE 2001: 282-293

Lima, G. et al. A Delta Oriented Approach to the Evolution and Reconciliation of Enterprise Software Products Lines. Iceis. Angers, p. 255-263. jul. 2013.

Lima, G. A. F. Uma Abordagem para Evolução e Reconciliação de Linhas de Produtos de Software Clonadas. Natal, RN: 2014. 184 f. Tese (Doutorado) -Universidade Federal do Rio Grande do Norte. Centro de Tecnologia. Programa de Pós-Graduação em Engenharia Elétrica e de Computação.

Neves, L. et al. Investigating the safe evolution of software product lines, Proceedings of the 10th ACM international conference on Generative programming and component engineering. 2011.

Nunes, C. et al. History-Sensitive Heuristics for Recovery of Features in Code of Evolving Program Families. SLP'12. 2012.

NUNES, Ingrid Oliveira de. Linhas de Produto de Software: Rio de Janeiro: Puc-rio, 2010. 72 slides, color. Disponível em: <<http://slideplayer.com.br/slide/344548/>>. Acesso em: 10 ago. 2015.

ORACLE AND/OR ITS AFFILIATES. Class TreeMap. 2014. Disponível em: <<http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html>>. Acesso em: 22 jul. 2015

Rubin, J., et al. 2012. Managing Forked Product Variants. Proceedings of the Software Product Line Conference (SPLC 2012). Salvador. Brazil.

Rubin, J; Czarnecki, K; Chechik, M. Managing Cloned Variants: A Framework and Experience Proceedings of the Software Product Line Conference (SPLC 2013). Tokyo, Japão.

Santos, J., et al. 2012. Conditional Execution: A Pattern for the Implementation of Fine-Grained Variabilities in Software Product Lines. Proceedings of 9th Latin- American Conference on Pattern Languages of Programming (SugarLoafPLoP 2012)

Santos, J., et al. 2015. MergeClear. An academic tool to deal with the problem of merge cloned systems. <https://github.com/jadsonjs/MergeClear> (last visited on August 27, 2015)

SANTOS, Pablo. Three-Way Merging: A Look Under the Hood. 2013. Disponível em: <<http://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902>>. Acesso em: 1 ago. 2015.

Sena, D., et al. 2012. Modularization of Variabilities from Web Information Systems Software Product Lines. Proceedings of 9th Latin-American Conference on Pattern Languages of Programming (SugarLoafPLoP 2012).

SIGAA– Sistema Integrado de Gestão de Atividades Acadêmicas. Superintendência de Informática da UFRN. Disponível em: http://www.info.ufrn.br/wikisistemas/doku.php?id=suporte:sigaa:visao_geral, Jun 2014.

Soares, A., et al. Programação Orientada a Aspectos. Uma visão geral. Departamento de Ciência da Computação. Universidade Federal de Lavras. Disponível em: <http://www.ic.unicamp.br/~rocha/college/src/aop.pdf>, Nov 2015

Van der Linden, F. J. Schmid, K. Rommes, E. Software Product Lines in Action. The Best Industrial Practice in Product Line Engineering. New York: Springer-verlag Berlin Heidelberg, 2007. 333 p. ISBN 978-3-540-71436-1

T. Mens, A State-of-the-Art Survey on Software Merging, IEEE Transactions on Software Engineering, v.28 n.5, p.449-462, May 2002
[doi>10.1109/TSE.2002.1000449]

Wang, X., Eclipse JDT tutorial – Java Search Engine – A Simple Working Example. Disponível em: <http://www.programcreek.com/2012/02/eclipse-jdt-tutorial-java-search-engine-example/>, Dez 2014.

Apêndice I: Tarefas Analisadas no Estudo

Este apêndice apresenta as tarefas que foram utilizadas para realizar a avaliação qualitativa desta dissertação.

Tabela I-1: Tarefas Analisadas para a Primeira Questão de Pesquisa

Nº	Descrição da Tarefa	Quantidade	Módulos
1	62263 - Reduzir passos de visualizar avaliacoes de discentes ead	7	false
2	67058 - SIGAA → Ensino a Distancia → Tutoria → Cadastro de Tutores Presenciais → Listar/Alterar	5	false
3	67798 - Dropar colunas de projeto_pesquisa que subiram para projeto	0	false
4	71138 - Habilitar funcao 'Recuperar acoes associadas excluidas'	4	true
5	71884 - Projetos de Apoio ao Novos Pesquisadores	16	true
6	26957 - Tarefas - Mostrar lista de alunos que acessaram as tarefas	4	false
7	31743 - Ref. matriculas em estagio e Atividades Complementares	16	false
8	39788 - ENVIAR E-MAIL AOS ALUNOS AVISANDO NOVA NOTICIA NO CURSO	7	true
9	73526 - Erro no fluxo e campos do caso de uso: Comite Interno → Comite Integrado de Ensino, Pesquisa e Extensao → Cadastrar.	6	true
10	61192 - Erro ortografico em Consultar Indices Academicos	2	false
11	81883 - Melhorias no cadastro de docente externo	10	true
12	68883 - NEE -> Relatorios/Consultas -> Consultas Gerais -> Orientacao de Atividades.	4	true
13	86119 - Verificacao nos relatorios das turmas nao consolidadas	1	false

14	87736 - Relatorios de Crescimento devem contar materiais excluidos	14	false
15	56380 - Verificar caso de uso: Comite de Extensao >> Membros do Comite >> Cadastrar Membro da Comissao	8	true

Tabela I-2: Tarefas Analisadas para a Segunda Questão de Pesquisa

Nº	Descrição da Tarefa	Quantidade	Módulos
1	57975 - Exibir tdas as subturmas na hora de consolidar	2	true
2	87467 - periodo de trancamento de modulos	1	true
3	88361 - alunos que desaparecem	1	true
4	47140 - Nomear Grupos	12	false
5	67401 - Solicitacoes Lato Sensu	1	true
6	88512 - 009 - Matricula de ingressantes	1	true
7	88001 - Turma nao consolida quando ja existe uma matricula consolidada	2	false
8	69824 - Turmas de Ensino Individualizado nao sao criadas com os alunos que as solicitou	2	true
9	95931 - Excluir trancamento de programa indevido	1	true
10	98327 - Ensino Individual	2	true

Tabela I-3: Tarefas Analisadas para a Terceira Questão de Pesquisa

Nº	Descrição da Tarefa	Quantidade	Nível de	Diferentes
1	51132 - VALIDACAO DE CONSOLIDACAO	1	2	false
2	72552 - Persistir Entidade StatusInscricaoSelecao	62	1, 2 e 3	true
3	80445 - Declaracoes de Membros de Grupo de Pesquisa	3	2, 3	true

4	87467 - periodo de trancamento de modulos	2	1	true
5	90357 - Reposicao de avaliacao	2	1 e 2	true
6	90563 - Erro - lancar notas	1	2	false
7	71885 - Projeto de Apoio aos Grupos de Pesquisa	20	2 e 3	false
8	61489 - Registrar data de aula extra na listagem de aulas para cadastro de topicos	1	2	true
9	63696 - Acao de extensao - inscricoes	2	1 e 2	true
10	61549 - Procedimento para corrigir informacoes dos alunos	5	1, 2, e 3	true
11	89582 - Gestor Avaliacao nao consegue visualizar o relatorio analitico	9	1, 2 e 3	true
12	67250 - Erro na criacao da turma	2	1 e 2	true
13	89380 - Servicos de Informacao e Referencia	14	1 e 2	false
14	80486 - Consolidar turma	2	2 e 3	true
15	92450 - corrigir dados de Certificados e Declaracoes	2	1 e 2	true

Apêndice II: Análise Quantitativa da Evolução Utilizada no Estudo.

Este apêndice apresenta alguns dados quantitativos levantados pela ferramenta sobre a evolução estuda (SIGAA UFRN 3.6.0 a 3.8.0, SIGAA UFOPA 3.6.0 a 3.7.64).

II.1 Tipos de Evolução Atômicas

As Figuras II-1 e II-2 mostram a quantidade de evoluções por tipo de artefato e tipo de modificação nos sistemas *Source* e no *Target* respectivamente.

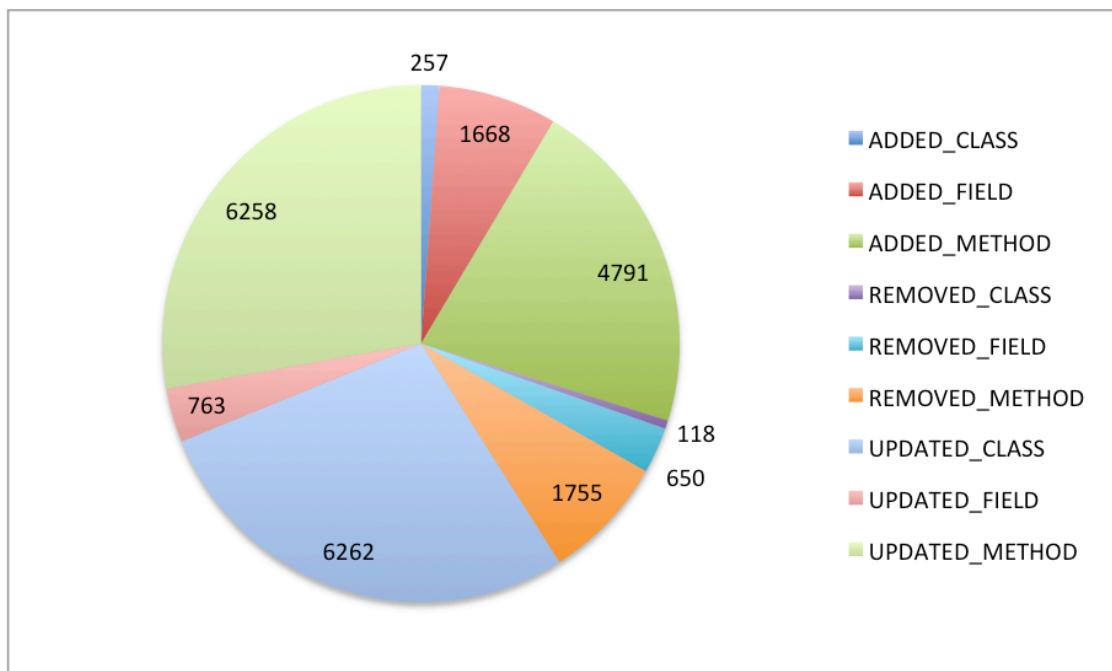


Figura II-1:Tipos de Evolução Atômicas do Source

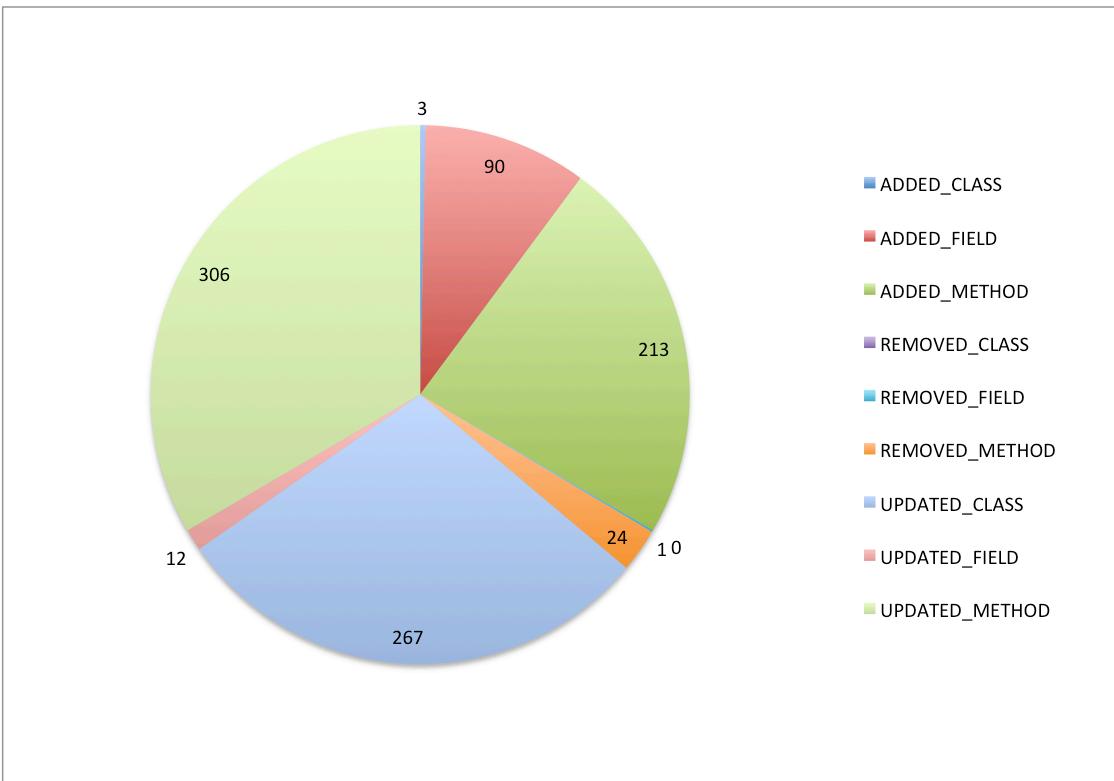


Figura II-2: Tipos de Evolução Atômicas do Target

II.2 Percentual de Conflitos por Tipos de Tarefas

As Figuras II-3 e II-4 mostram a porcentagem de conflitos por tipo de tarefa nas evoluções dos sistemas *Source* e no *Target* respectivamente.

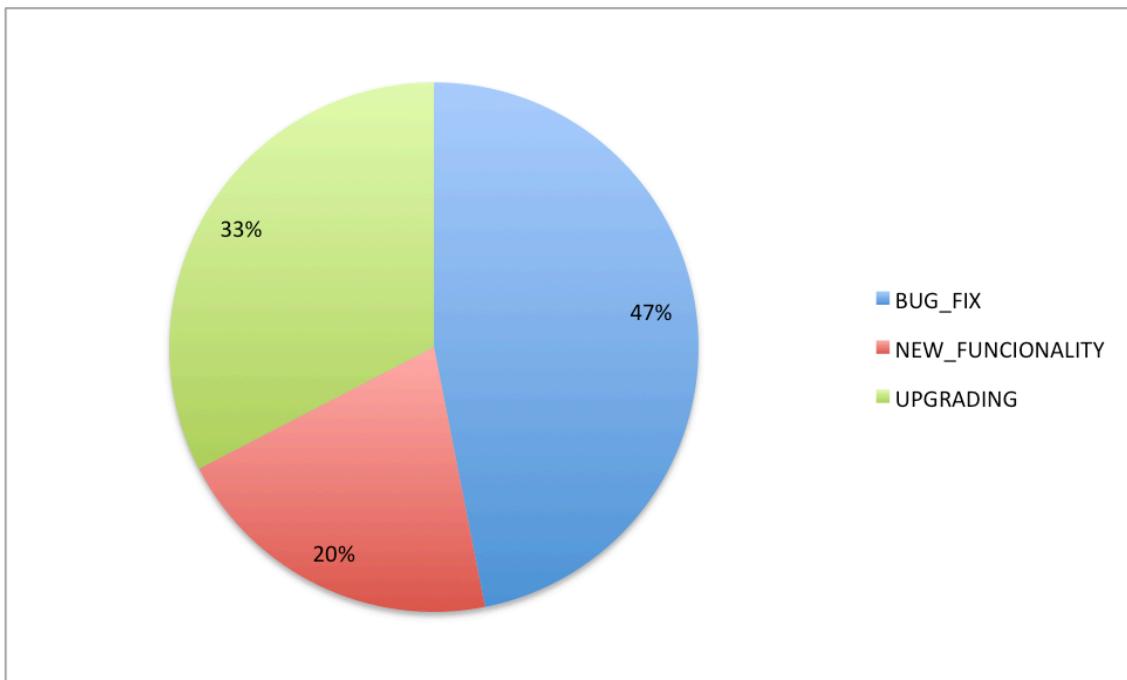


Figura II-3:Tipos de Tarefas no Source

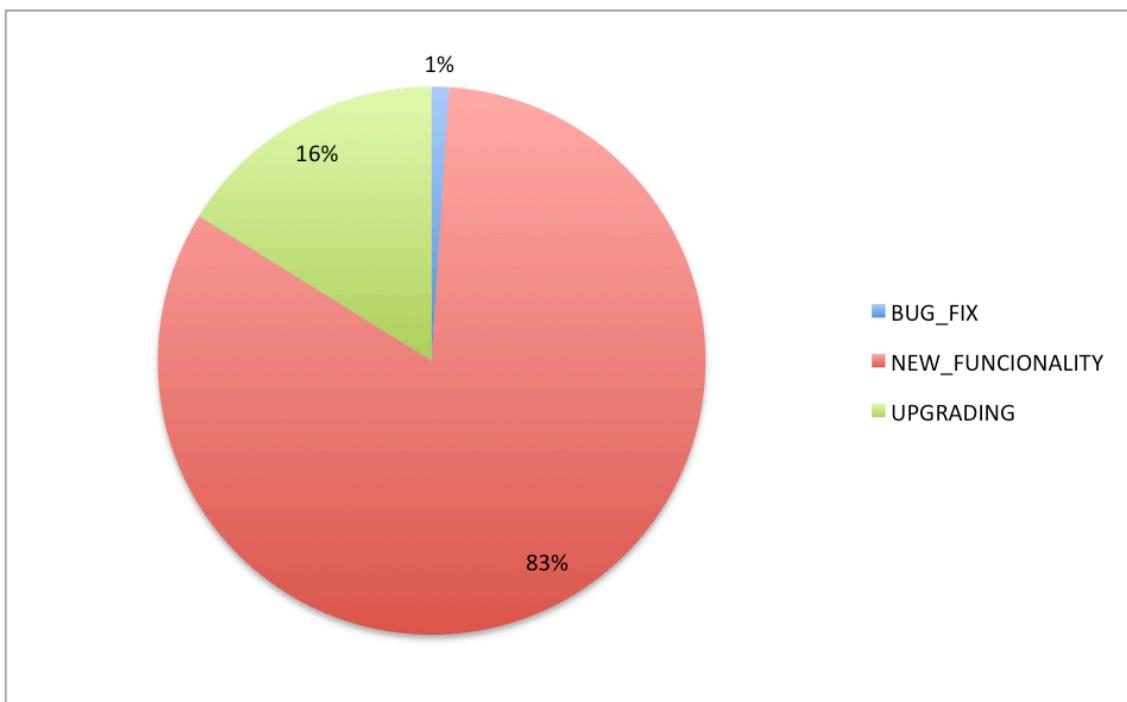


Figura II-4:Tipos de Tarefas no Target

II.3 Percentual de Tipos de Conflitos

As Figuras II-5 e II-6 mostram a porcentagem dos tipo de conflitos que foram detectados nas evoluções dos sistemas *Source* e no *Target* respectivamente.

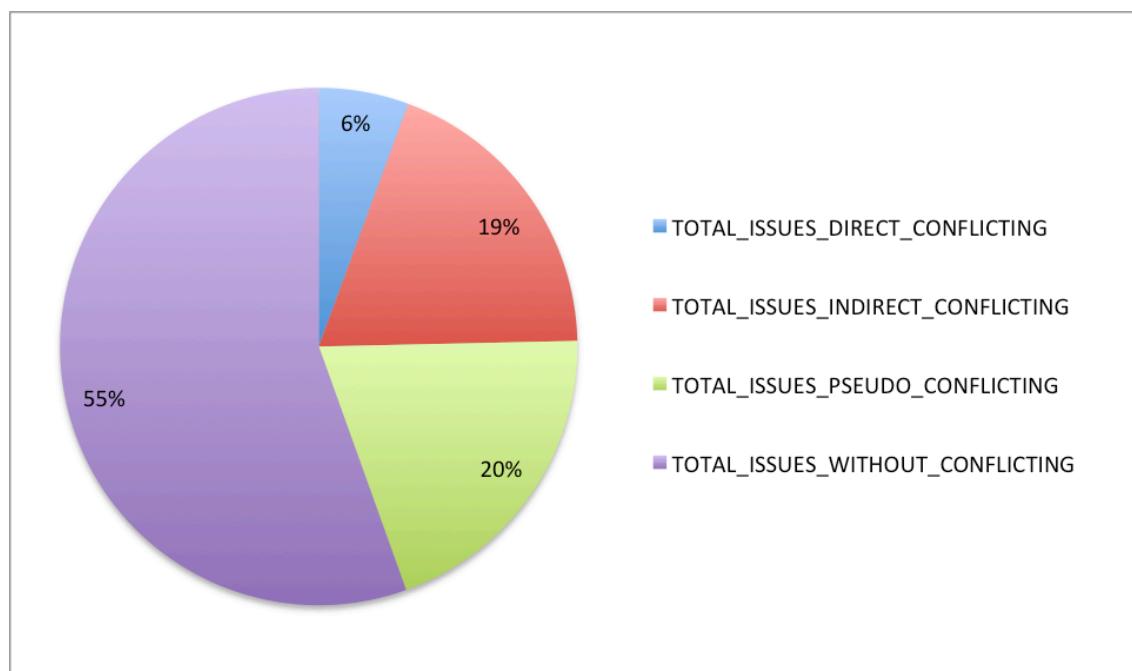


Figura II-5: Porcentagem dos Tipos de Conflitos no Source

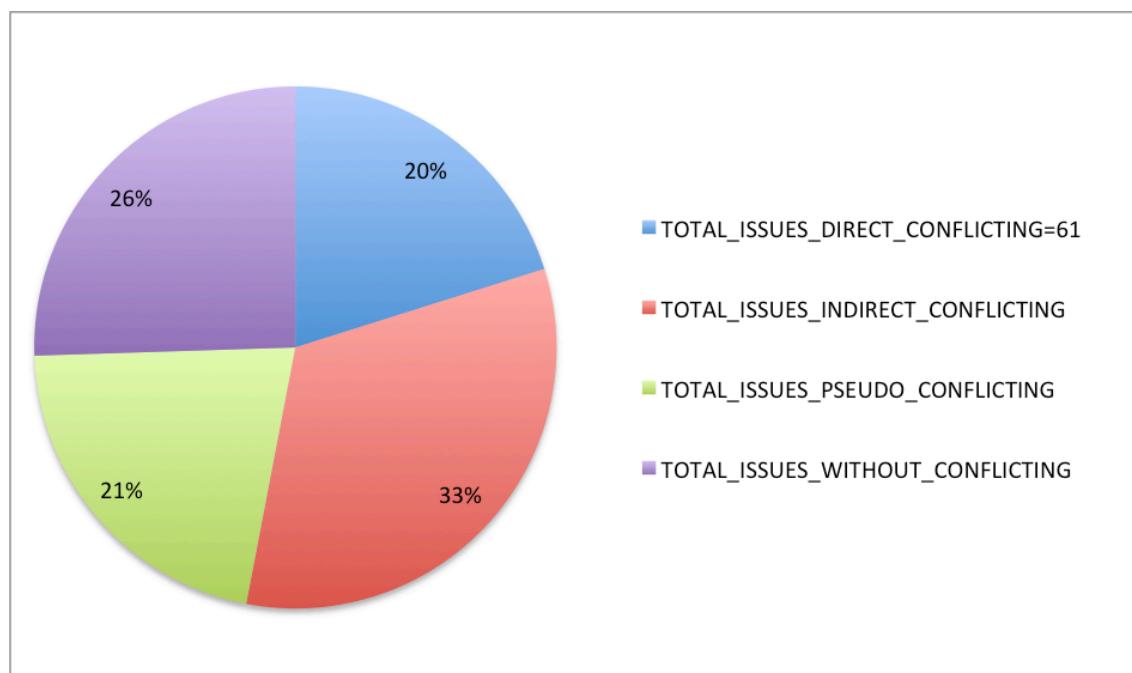


Figura II-6: Porcentagem dos Tipos de Conflitos no Target

II.4 Conflitos por Tipos de Evolução

As Figuras II-7 e II-8 mostram a quantidade dos tipo de conflitos por tipo de tarefa que foram detectados nas evoluções dos sistemas *Source* e no *Target* respectivamente.

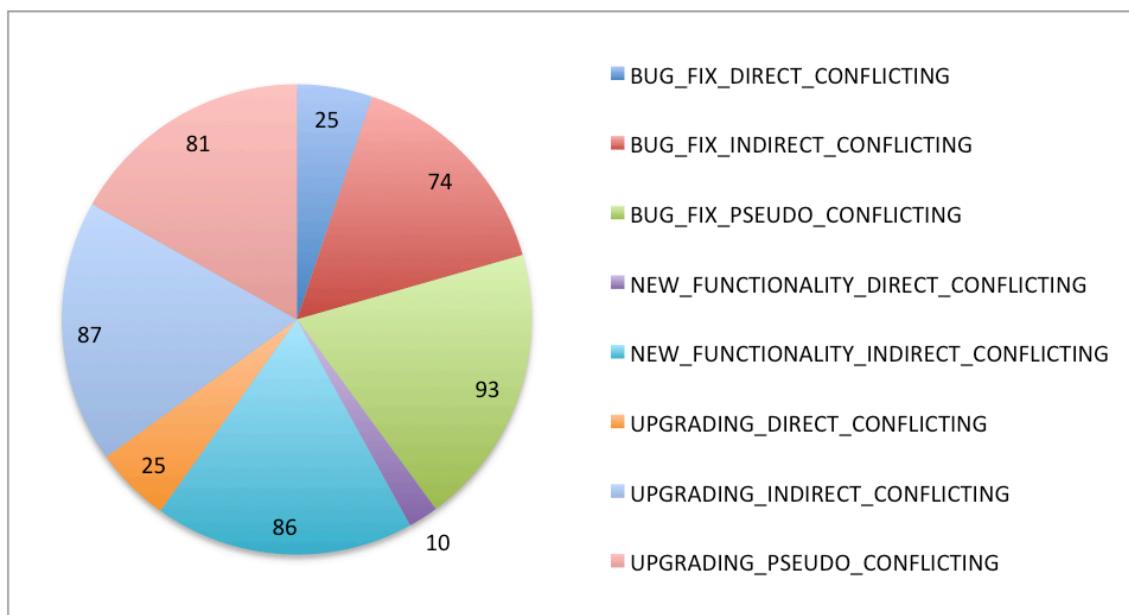


Figura II-7: Quantidade de Conflitos por Tipo de Tarefa no Source

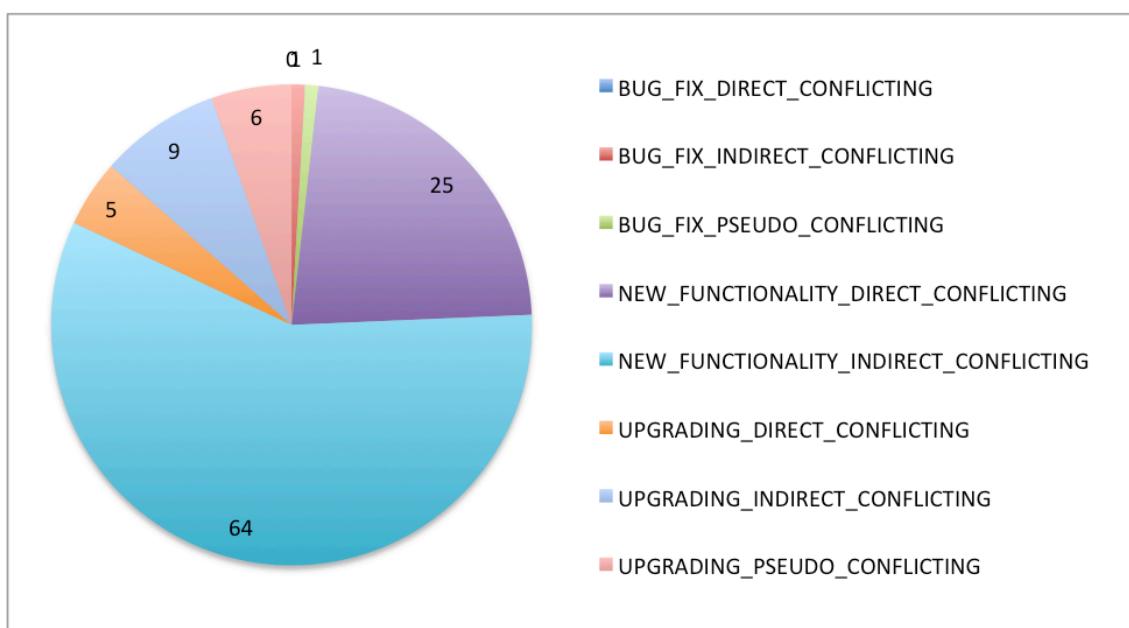


Figura II-8: Quantidade de Conflitos por Tipo de Tarefa no Target

II.5 Tipos de Artefatos por Tipo de Conflito

As Figuras II-9 e II-10 mostram a quantidade de artefatos que sofreram evolução por tipo de conflito que foram detectados pela ferramenta nos sistemas *Source* e no *Target* respectivamente.

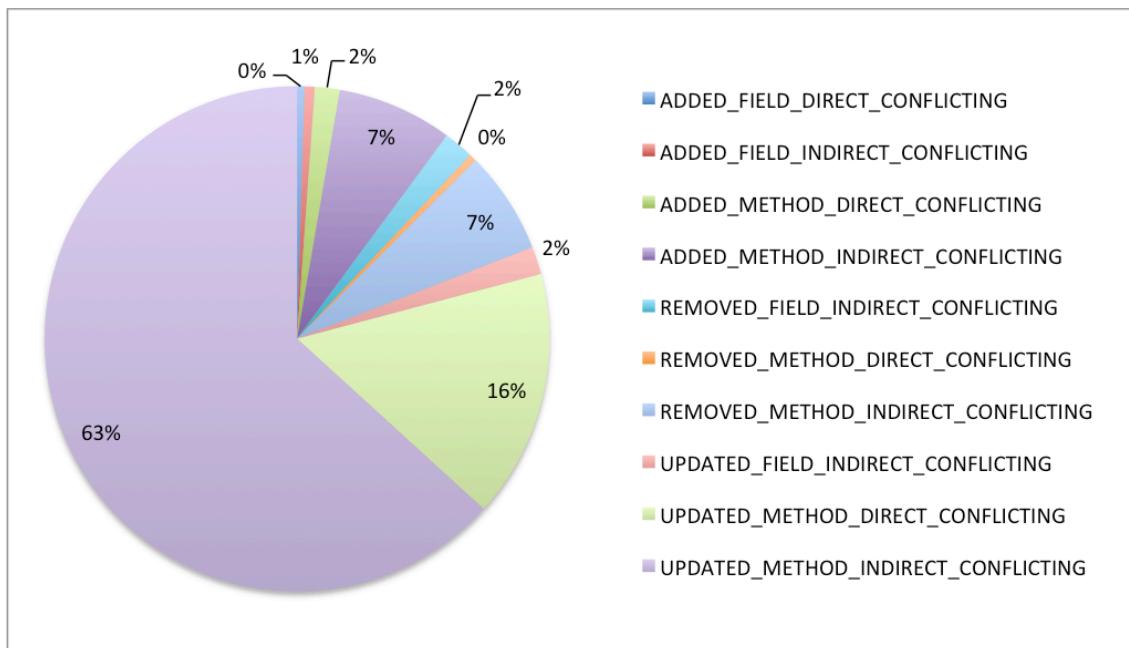


Figura II-9: Tipos de Conflitos por Tipos de Artefatos no *Source*

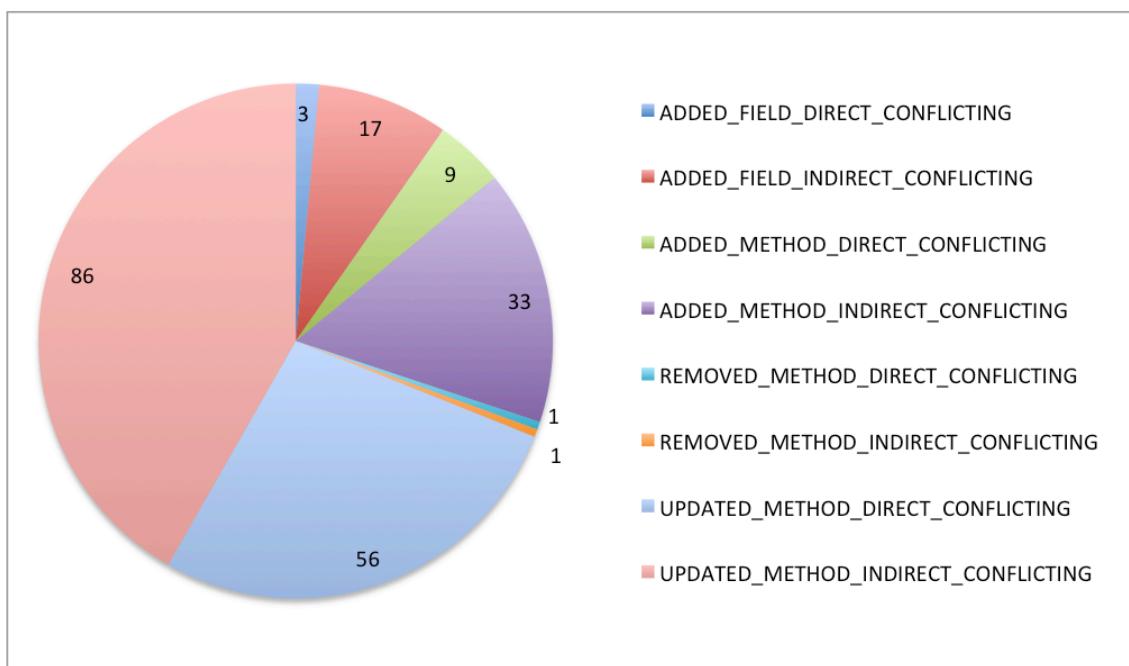


Figura II-10: Tipos de Conflitos por Tipos de Artefatos no *Target*