

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



**TÉCNICAS PARA COMPREENSÃO DE RASTROS DE
EXECUÇÃO DE PROGRAMAS ORIENTADOS A OBJETOS**

LUCIANA LOURDES SILVA

Uberlândia - Minas Gerais

2011

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



LUCIANA LOURDES SILVA

TÉCNICAS PARA COMPREENSÃO DE RASTROS DE EXECUÇÃO DE PROGRAMAS ORIENTADOS A OBJETOS

Dissertação de Mestrado apresentada à Faculdade de Ciência da Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Engenharia de Software.

Orientador:

Prof. Dr. Marcelo de Almeida Maia

Co-orientadora:

Prof^a. Dr^a. Sandra Aparecida de Amo

Uberlândia, Minas Gerais

2011

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Ciência da Computação a aceitação da dissertação intitulada “**Técnicas para Compreensão de Rastros de Execução de Programas Orientados a Objetos**” por **Luciana Lourdes Silva** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 22 de Fevereiro de 2011

Orientador:

Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Co-orientadora:

Prof^a. Dr^a. Sandra Aparecida de Amo
Universidade Federal de Uberlândia

Banca Examinadora:

Prof. Dr. Michel dos Santos Soares
Universidade Federal de Uberlândia

Prof. Dr. Marco Túlio de Oliveira Valente
Universidade Federal de Minas Gerais

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Fevereiro de 2011

Autor: **Luciana Lourdes Silva**
Título: **Técnicas para Compreensão de Rastros de Execução de Programas Orientados a Objetos**
Faculdade: **Faculdade de Ciência da Computação**
Grau: **Mestrado**

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão de cópias deste documento para propósitos exclusivamente acadêmicos, desde que o autor seja devidamente informado.

Autor

O AUTOR RESERVA PARA SI QUALQUER OUTRO DIREITO DE PUBLICAÇÃO DESTE DOCUMENTO, NÃO PODENDO O MESMO SER IMPRESSO OU REPRODUZIDO, SEJA NA TOTALIDADE OU EM PARTES, SEM A PERMISSÃO ESCRITA DO AUTOR.

Dedicatória

Ao meu noivo Jânio Rosa.

A minha mãe Zilda e a minha irmã Leide.

Ao meu pai honorário Edson e a minha tia Rosalina.

Agradecimentos

Primeiro eu gostaria de agradecer a Deus pela minha vida. Segundo, pelas amizades sinceras que conquistei e os bons resultados que obtive deste trabalho durante o período do mestrado.

Eu gostaria de agradecer meu orientador, o Prof. Dr. Marcelo Maia, pelo profissionalismo, paciência e presença em todos os momentos para a concretização deste trabalho. Seu entusiasmo, dedicação e amplo conhecimento me ajudaram a obter uma experiência considerável durante o tempo que trabalhamos juntos.

Eu também gostaria de agradecer minha co-orientadora, a Prof.^a Dr.^a Sandra de Amo, pelos *feedback* que com sua vasta experiência nos ajudou na elaboração de artigos e pelo reconhecimento do nosso trabalho a cada artigo aceito, que aumentou muito minha motivação pela continuidade.

Eu quero agradecer em especial o meu noivo Jânio, pelas dicas de implementação e por sempre acreditar na minha capacidade, me dando apoio e motivação.

Eu também quero agradecer minha mãe Zilda e minha irmã Leide pelo apoio e compreensão da minha ausência em muitos momentos. Em particular, quero agradecer ao meu pai honorário Edson e a minha tia Rosalina, que tiveram uma participação importante na minha formação.

Finalmente, obrigada aos meus amigos de laboratório, Nádia Félix e Allysson Costa pelo encorajamento e motivação, que de certa maneira contribuíram para a realização deste trabalho.

*“Quem sabe concentrar-se numa coisa e insistir nela como único objetivo, obtém, ao fim
e ao cabo, a capacidade de fazer qualquer coisa.”
(Mahatma Gandhi)*

Resumo

Várias abordagens para facilitar a compreensão do comportamento de sistemas de software têm sido propostas. Mudanças perfectivas em sistemas de *software* bem estabelecidos são mais fáceis de executar quando a equipe de desenvolvimento tem um entendimento sólido do código fonte. Mas é razoável assumir que o uso de um sistema de código aberto para incorporar novas características e obter um novo produto de *software* é uma abordagem interessante, ao invés de codificar um novo produto a partir do zero. Em consideração a este cenário e considerando que não é incomum sistemas pobres em documentação, não existe uma abordagem amplamente aceita para guiar em mudanças perfectivas desenvolvedores com baixo conhecimento do sistema ou que recupera informações em alto nível de abstração sobre a estrutura e comportamento de sistemas complexos.

Este trabalho propõe uma nova abordagem para simplificar tarefas de compreensão de programas orientados a objetos através da análise de rastros de execução sumarizados. A abordagem é aplicada sobre duas técnicas: a primeira permite a separação de partes comuns do código fonte das partes específicas relacionadas a características importantes que conduz a adição de uma nova. Uma avaliação é feita para verificar se os rastros de execução sumarizados ajudam a técnica na localização de elementos potenciais de código que podem guiar o desenvolvimento de uma nova característica. A avaliação foi realizada com sistemas do mundo real e com tarefas de evolução significativas. A segunda é baseada na reconstrução de diagramas estruturais e comportamentais de alto nível baseada na análise de rastros de execução sumarizados. É apresentada uma avaliação do desempenho da abordagem em termos de precisão e *recall* em dois sistemas públicos de terceiros, dentre eles o servidor Web Tomcat. O resultado sugere a viabilidade da abordagem para uso em sistemas reais de larga escala.

Palavras chave: evolução de *software*, rastros de execução, reuso, manutenção de *software*, sumarização de rastros, alinhamento de rastros, visões dinâmicas

Abstract

Several attempts to facilitate understanding the behavior of software systems have been proposed. Perfective changes in well-established software systems are easier to perform when the development team has a solid understanding of the internals. However, it is reasonable to assume that the use of an open source system to incorporate new features and obtain a new software product is an appealing approach instead of coding a new product from scratch. Considering this scenario, and considering that it is not uncommon that systems are poorly documented, there is no widely accepted approach to guide the perfective maintenance for developers with low understanding of the system or that recovers high-level information about both the structure and the behavior of large systems.

This work proposes a new approach to simplify comprehension tasks of object oriented programs through the analysis of summarized execution traces. The approach is performed on two techniques: The first technique enables the separation of common parts of source code from specific parts related to important features that drive the addition of the new one. An evaluation is done to verify if the summarized execution traces helps the technique to locate potential elements of code that can guide the development of a new feature. The evaluation was conducted with real-world systems and with meaningful evolution tasks. The second is based on a technique that reconstructs structural and behavioral high-level diagrams by the analysis of summarized execution traces. Precision and recall were evaluated using two third-party open-source systems, including the webserver Tomcat. The result suggests the feasibility for using the approach on real world large scale systems.

Keywords: software evolution, execution traces, reuse, software maintenance, trace summarization, trace alignment, dynamic views

Sumário

Lista de Figuras	xi
Lista de Tabelas	xiii
1 Introdução	14
2 Coleta e Sumarização de Rastros de Execução	21
2.1 Instrumentação do Sistema e Execução dos Cenários de Execução	22
2.2 Algoritmo de Sumarização dos Rastros	23
2.2.1 Análise do Algoritmo	25
2.2.2 Representação dos Rastros em uma <i>Tree View</i>	26
2.3 Separação por Tarefas	27
2.4 Considerações Finais	28
3 Alinhamento de Rastros de Execução Sumarizados	29
3.1 Algoritmo de Alinhamento de Rastros de Execução	29
3.2 Processo de Alinhamento de Rastros Sumarizados	31
3.2.1 Definição dos Cenários de Execução	31
3.2.2 Alinhamento dos Rastros de Execução Sumarizados	33
3.2.3 Definição do Plano de Implementação	34
3.3 Estudos de Caso e Avaliação dos Resultados	35
3.3.1 Estudo de Caso 1 - ArgoUML	35
3.3.2 Estudo de Caso 2 - Columba	41
3.3.3 Estudo de Caso 3 - Llama Chat	45
3.4 Discussão	48
3.5 Considerações Finais	49
4 Extração de Visões de Alto Nível de Abstração	50
4.1 Definição dos Cenários de Execução	50
4.2 Análise dos Rastros Sumarizados	52
4.3 Agrupamento da <i>Tree View</i>	55
4.4 Estudos de Caso e Avaliação dos Resultados	56

4.4.1	Avaliação dos Resultados	56
4.4.2	Análise do Sistema de Simulação ATM	57
4.4.3	Análise do Tomcat	60
4.5	Considerações Finais	64
5	Trabalhos Correlatos	65
5.1	Sumarização de Rastros de Execução	65
5.2	Compreensão de Sistemas de <i>Software</i>	69
5.3	Considerações Finais	72
6	Conclusão e Trabalhos Futuros	73
	Referências Bibliográficas	77

Lista de Figuras

2.1	Principais passos do processo de sumarização de rastros.	22
2.2	Exemplo de compressão dos rastros de execução.	23
2.3	Algoritmo <i>Summarizer</i> com o parâmetro da granularidade mínima. . . .	24
2.4	Algoritmo do método Encontra_Granularidade	24
2.5	Algoritmo Sumariza_Rastro com o parâmetro da granularidade mínima.	24
2.6	Algoritmo <i>Summarizer</i> automatizado.	25
2.7	Fragmento de rastro de execução sumarizado da inicialização do ArgoUML.	26
2.8	(a) mostra um rastro comprimido, onde os laços e recursões foram removi- dos e (b) mostra o rastro após a execução do algoritmo <i>Summarizer</i>	27
2.9	Exemplo de repositório de rastros após a execução da separação por tarefas.	28
3.1	Exemplo de alinhamento entre duas sequências de rastros de execução. . .	30
3.2	Descrição das 4 fases da técnica de alinhamento.	31
3.3	A abordagem de alinhamento de rastros sumarizados.	33
3.4	Fragmento da árvore que contém a inicialização do ArgoUml.	37
3.5	Fragmento da árvore que ilustra a sequência de chamada específica para cada cenário.	38
3.6	(a) Fragmento da árvore de alinhamento para detectar as chamadas de métodos para criar um botão na barra de ferramentas e (b) fragmento da árvore que contém desalinhamento.	43
3.7	Segmento desalinhado referente ao recebimento de mensagem do Llama Chat.	47
4.1	Descrição das 4 fases da técnica para recuperar visões de alto nível sobre rastros sumarizados.	51
4.2	Árvore simplificada para extração de informação.	52
4.3	Transformação da <i>Tree View</i> para o diagrama de sequência.	53
4.4	Transformação da <i>Tree View</i> para o diagrama de pacotes do Javac. . . .	54
4.5	Fases de agrupamento do algoritmo.	55
4.6	Diagrama de sequência de Consulta de Saldo do Simulador ATM e seus componentes encontrados.	59

4.7	Componentes detectados pelo algoritmo de agrupamento da <i>Tree View</i> do Tomcat.	62
-----	---	----

Lista de Tabelas

2.1	Par de Cenários para o ArgoUML	23
3.1	Exemplo de cenários para detectar código comum e específico que implementam as características.	32
3.2	Par de Cenários para o ArgoUML	36
3.3	Resultados obtidos da execução dos algoritmos de Compressão e Sumarização dos Rastros de Execução do ArgoUML.	37
3.4	Número de elementos inseridos no ArgoUML.	41
3.5	Precisão e <i>Recall</i> no ArgoUML.	41
3.6	Par de cenários para atualizar o Columba.	41
3.7	Par de cenários para reusar código existente no Columba.	42
3.8	Resultados da Compressão e Sumarização dos Rastros de Execução do Columba.	42
3.9	Conjunto de métodos do plano de implementação para a tarefa enviar mensagem no Columba.	44
3.10	Número de elementos inseridos no Columba.	45
3.11	Precisão e <i>Recall</i> no Columba.	45
3.12	Par de cenários para o Llama Chat.	46
3.13	Resultados da Compressão e Sumarização dos Rastros de Execução do Llama Chat.	46
3.14	Precisão e <i>Recall</i> no Llama Chat.	48
4.1	Cenário de Execução para o <i>Prevayler</i>	51
4.2	Cenário de Execução para o Simulador ATM	57
4.3	Resultados da Compressão e Sumarização dos Rastros de Execução do Simulador ATM.	57
4.4	Precisão e <i>Recall</i> para as duas características do simulador ATM.	60
4.5	Resultados da Compressão e Sumarização dos Rastros do Tomcat	60
4.6	Precisão e <i>Recall</i> para as duas tarefas do Tomcat.	63

Capítulo 1

Introdução

A quantidade de código aberto disponível na Internet fornece para várias equipes de desenvolvimento código fonte útil que pode ser incorporado em seus produtos. No entanto, é sabido também que quase todos estes sistemas não possuem documentação adequada ou quando possuem não é garantido que estão atualizadas ou completas [Lethbridge et al. 2003].

Sem o auxílio de ferramentas eficientes, a análise de um sistema complexo torna-se extremamente difícil. Isto é porque existe uma tendência de perder a rastreabilidade dos artefatos do sistema e os relacionamentos entre eles [Hamou-lhadj e Lethbridge 2004].

O custo de manter um sistema de *software* e gerenciar sua evolução representa mais de 90% do custo total de um projeto de *software* [Seacord et al. 2003].

Manutenção de *software* é definido como a modificação de um produto de *software* após a entrega para corrigir falhas, melhorar o desempenho ou outros atributos, ou adaptar o produto a um ambiente modificado, [Subcommittee 1992], [Bennett e Rajlich 2000]. Estas modificações são geralmente classificadas em quatro categorias:

Manutenção Adaptativa. Modificação de um sistema existente para adaptar a novos requisitos.

Manutenção Corretiva. Modificação de um sistema existente para corrigir falhas encontradas.

Manutenção Preventiva. Está relacionada com prevenir falhas antes mesmo delas ocorrerem. Por exemplo, um engenheiro de *software* talvez adicione detectores de falhas se a possibilidade de uma falha potencial seja notada.

Manutenção Perfectiva. Modificar sistemas existentes para melhorar o desempenho, atender pedidos do usuário para modificar e incluir novas características.

A maior parte do esforço em modificar um sistema de *software* para adicionar uma nova característica baseia-se na compreensão dos requisitos e artefatos de código. Um dos problemas mais significativos em experiências atuais de evolução de *software* é a dificuldade do mantenedor encontrar partes no código fonte onde novas características foram inseridas ou onde mudanças devem ser feitas. A rastreabilidade entre requisitos e artefatos de código é ainda um grande desafio na manutenção perfectiva [Lucia et al. 2009].

Existem técnicas que auxiliam a identificar características no código fonte, tanto por meio da análise dinâmica quanto pela análise estática. Análise dinâmica é a análise das propriedades de um programa em tempo de execução, que são derivadas a partir de uma ou mais execuções de um programa instrumentado. Diferente da análise dinâmica, a análise estática estuda o código do sistema sobre todos os possíveis comportamentos que talvez surjam em tempo de execução [Mit e Ernst. 2003].

Precisão de informação. Usar análise dinâmica em um contexto de engenharia ou reengenharia de *software* requer a geração de um rastro de execução do sistema alvo. Os rastros de execução são obtidos através da execução de um cenário bem definido no sistema instrumentado. O cenário é uma instância de um ou vários casos de uso [Jacobson 1995]. Os rastros de execução contêm precisamente todas as chamadas de métodos, no caso de sistemas orientados a objetos, que não são bibliotecas e que foram necessárias para executar um determinado cenário [Ball 1999]. No entanto, existem algumas desvantagens da análise dinâmica, como por exemplo a escalabilidade, por causa dos rastros de execução extremamente volumosos que são produzidos, afetando o desempenho e o espaço de armazenamento. Esta escalabilidade encontra limites computacionais relacionados à complexidade de tempo e espaço dos algoritmos, e os resultados que são apresentados aos usuários são muito grandes tornando difíceis de serem compreendidos [Zaidman 2006].

A compreensão de programas é uma atividade central durante a manutenção de *software*, evolução e reúso. Estima-se que mais de 50% do esforço de manutenção é gasto na tentativa de compreender o código fonte [Corbi 1989], [von Mayrhauser e Vans 1995]. Este esforço pode ser reduzido significativamente através de técnicas que fornecem visões com o objetivo de simplificar a compreensão [Clements et al. 2002]. Compreensão de programas envolve geralmente estratégias também referidas como modelos de compreensão de programas. Duas das principais são descritas resumidamente:

Modelo *Bottom-up*. A compreensão do código fonte é feita a partir da construção de abstrações dele. Esta estratégia geralmente envolve ler o código fonte e agrupar detalhes de programação de baixo nível em níveis mais alto, obtendo estruturas semanticamente mais significativas.

O processo de agrupar blocos pequenos de informação em blocos maiores é chamado

de *chunking*. Este processo é repetido até que uma compreensão adequada do sistema seja adquirida [Pennington 1987], [von Mayrhauser e Vans 1995].

Um rastro de execução pode ser representado em uma árvore. A compreensão *bottom-up* de um rastro consiste em explorar o conteúdo de várias sub-árvores do rastro. Esta exploração geralmente está relacionada com expandir e ocultar sub-árvores, como é apresentado em [Jerding e Rugaber 1997].

Modelo *Top-Down* [Soloway e Ehrlich 1984], [Soloway et al. 1988]. A compreensão de programas começa com um objetivo de alto nível e prossegue com a geração de sub-objetivos detalhados que são necessários para executar os objetivos de mais alto nível. [von Mayrhauser e Vans 1995] diz que este modelo aplica-se quando o código ou tipo de código é familiar. Por exemplo: suponha que seja solicitado a um desenvolvedor experiente, com especialidade em sistemas operacionais, para manter um sistema operacional que ele não conheça. Por ser experiente, ele pode imediatamente decompor o novo sistema em elementos que ele sabe que deve estar implementado no código: um gerenciador de processos, um gerenciador de arquivos, um gerenciador de I/O e um gerenciador de memória. Cada um destes elementos podem ser decompostos, por exemplo, o gerenciamento de processos inclui comunicação entre processos e escalonamento de processos. Escalonamento de processos pode ser implementado usando por exemplo o algoritmo de escalonamento de prioridade. O desenvolvedor talvez continue esta análise *top down* até que reconheça um bloco de código, que neste caso é o algoritmo de escalonamento de processos. Durante a compreensão, não é necessário reaprender este algoritmo linha por linha. O desenvolvedor só precisa reconhecer que o código apropriado existe. Teoricamente, o novo código poderia ser compreendido por completo de uma maneira *top down* se o desenvolvedor já tenha se deparado com a mesma tarefa e o código estruturado da mesma maneira.

Várias tentativas para facilitar a compreensão de sistemas de software surgiram baseadas nas hipóteses que o código fonte em si é a única fonte de informação confiável sobre o sistema disponível [Müller et al. 1993], [Hamou-lhadj e Lethbridge 2002], [Lethbridge et al. 2003], [Ducasse e Pollet 2009]. Mas, técnicas que auxiliem na compreensão de programas são particularmente insuficientes para habilitarem desenvolvedores a compreenderem de maneira rápida a implementação de uma característica do *software*. IDEs recentes fornecem ferramentas de depuração poderosas para compreender o comportamento do sistema, mas fornecem somente visões simplificadas sobre a arquitetura que são definidas principalmente pela organização de pacotes do projeto.

Atualmente, abordagens tradicionais para realizar tarefas de evolução, que inserem novas características no sistema, incluem na compreensão do sistema por meio de análise da documentação e/ou depuração de características semelhantes como uma alternativa inicial para compreender o sistema. Uma das hipóteses deste trabalho é que a inserção

de uma nova característica pode ser executada com baixo esforço por analisar artefatos semelhantes existentes no sistema. A hipótese supõe que é possível descobrir onde os novos artefatos serão inseridos ou modificados e também a forma inicial destes artefatos.

Existem várias iniciativas para ajudar na compreensão de sistemas em alto níveis de abstração, como técnicas de recuperação de arquitetura [Yan et al. 2004], [Sartipi e Dezhkam 2007]. Estas abordagens também usam informações estáticas sobre o código ou usam informações dinâmicas de programas, como rastros de execução ou mesmo uma combinação de ambos [Mit e Ernst. 2003]. A compreensão da dinâmica de um sistema de *software* pode ser mais simples se técnicas de análise dinâmica forem usadas. Análise dinâmica geralmente envolve a análise dos rastros gerados da execução de características de um sistema alvo [Ball 1999]. Rastros de execução têm sido aplicados para identificar quais partes do código implementam algumas características específicas [Sobreira e Maia 2008]. No entanto, extrair informação importante dos rastros de execução é uma tarefa difícil, porque são extremamente volumosos podendo conter milhares ou até milhões de chamadas de métodos. Existem técnicas de simplificação de rastros que consistem na remoção de laços e recursões [Hamou-lhadj e Lethbridge 2002] porque, no geral, chamadas de métodos repetitivas não fornecem informação interessante para compreender a estrutura do sistema e seu benefício no entendimento da dinâmica do sistema parece não pagar o preço de ter um rastro de execução muito grande. Mesmo com estas remoções, os rastros geralmente ainda permanecem muito volumosos. Várias técnicas de redução de rastros de execução têm sido propostas com o objetivo de reduzir rastros muito volumosos para aumentar a precisão na busca de informação, [Pauw et al. 1998], [Reiss e Renieris 2001], [Hamou-Lhadj e Lethbridge 2006], [Kuhn e Greevy 2006], [Prada-Rojas et al. 2010]. No entanto, é preciso escolher a técnica de redução ideal para uma determinada situação. Existem abordagens que possibilitam análises de técnicas de redução de rastros para selecionar diferentes técnicas que são aplicadas em contextos específicos [Cornelissen et al. 2008]. Mesmo após a compressão e redução dos rastros, eles permanecem complexos para interpretar seu conteúdo. Assim, é necessária a aplicação de técnicas que transformam as informações dos rastros reduzidos em modelos comportamentais [Taniguchi et al. 2005], [Kuhn e Greevy 2006] ou estruturais [Reiss e Renieris 2001], [Sartipi e Dezhkam 2007] que auxiliem o analista ou desenvolvedor a compreender partes importantes de um sistema.

Objetivos

O objetivo deste trabalho é simplificar a tarefa de compreensão de programas para auxiliar em atividades de reuso e evolução de *software* através da aplicação de uma nova abordagem de sumarização de rastros de execução e de técnicas de recuperação de informação sobre rastros sumarizados. A abordagem de sumarização de rastros de execução

busca definir uma noção de chamadas de métodos relevantes. Os rastros sumarizados são utilizados para compreender características de um sistema sem precisar ler todo o seu conteúdo. O objetivo das técnicas aplicadas sobre os rastros sumarizados é capacitar a análise *top down*. A primeira extrai informação do código fonte que direciona na compreensão de sistemas para inserir novas características. A segunda técnica extrai informações com visões de alto e baixo nível de características específicas dos rastros sumarizados para auxiliar na redocumentação de sistemas existentes.

Organização e Proposta da Dissertação

Uma nova abordagem para análise de rastros de execução é apresentada neste trabalho através da redução de tamanho dos rastros. A técnica de sumarização de rastros é baseada na noção de chamadas de métodos relevantes, onde uma chamada de método é considerada relevante se o número de chamadas internas a ela é alto. A técnica busca remover chamadas de métodos com baixa granularidade. Assim, o problema de perda de informações importantes é balanceado, evitando poda excessiva que talvez elimine chamadas de métodos importantes para a compreensão.

Capítulo 2 - Coleta e Sumarização de Rastros de Execução. Este capítulo apresenta os principais passos do processo de sumarização de rastros. Primeiro, a instrumentação do sistema e a coleta dos rastros através da execução dos cenários. Segundo, uma nova abordagem de sumarização de rastros de execução baseada na noção de granularidade das chamadas de métodos. O algoritmo de sumarização elimina chamadas de métodos com baixa granularidade que é calculada sobre o número de chamadas que ocorreram dentro de uma chamada atual. Assume-se como hipótese que, chamadas de métodos com granularidades maiores em um rastro são os métodos mais relevantes para a execução de uma determinada característica.

Através dos rastros sumarizados é possível extrair diferentes tipos de informações dos rastros de execução por meio da aplicação de diferentes técnicas. Neste trabalho é realizado a aplicação de duas técnicas. A primeira é baseada no alinhamento de rastros de execução, proposta por [Paixão 2009], sobre os rastros sumarizados que permite separar partes comuns de código fonte das partes específicas relacionadas a características importantes que irão direcionar a adição de uma nova em um sistema alvo. Resultados preliminares mostraram que o resultado de alinhamento de rastros de execução podem resultar em uma quantidade de informação extremamente grande que talvez torne a abordagem inviável [Paixão 2009]. Neste trabalho é feito a combinação de compressão de rastros, proposta por [Hamou-lhadj e Lethbridge 2002], com a abordagem de sumarização para viabilizar o uso de alinhamento de rastros.

Capítulo 3 - Alinhamento de Rastros de Execução Sumarizados. Este capítulo apresenta o conceito de alinhamento de rastros de execução, de características semelhantes definindo os cenários de execução de características similares à nova a ser inserida em um sistema alvo. O capítulo também tem como objetivo mostrar a aplicação da técnica de alinhamento de rastro de execução sobre rastros sumarizados, recuperando informações importantes do código fonte que podem guiar na compreensão de sistemas para reduzir o esforço nas tarefas de evolução. Para este propósito, a abordagem de alinhamento de rastros é executada para encontrar elementos comuns e de variação no código fonte. Com o objetivo de avaliar a técnica de alinhamento sobre os rastros sumarizados, o processo de evolução dos sistemas de *software* ArgoUML, Llama Chat e Columba é apresentado por adicionar novas características semelhantes a características existentes nos respectivos sistemas e uma avaliação para cada estudo de caso da qualidade dos resultados obtidos a partir do alinhamento dos rastros sumarizados e representados em *Tree Views*.

A segunda técnica busca omitir detalhes de implementação que não são removidos pela aplicação da técnica de sumarização, por meio de filtros para extrair informações com visões de alto nível de abstração de características específicas do sistema. Os resultados obtidos pela técnica permitem elaborar diagramas de sequência e de pacotes de alto nível. Segundo [Delamare et al. 2006], gerar diagramas de sequência básicos, que são os diagramas com detalhes de implementação, é fácil, mas gerar diagramas de sequências de alto nível é uma tarefa complexa. As IDEs possuem mecanismos de depuração que auxiliam a compreender como os objetos se interagem uns com os outros, a cooperação do ponto de vista global não é clara, enquanto que esta técnica tem como objetivo fornecer visões estruturais e comportamentais de alto nível, sob o ponto de vista global de uma característica do sistema.

Capítulo 4 - Extração de Visões de Alto Nível de Abstração. Neste capítulo são apresentados algoritmos propostos para simplificar os rastros sumarizados de maneira que as informações são somente omitidas ao analista. Com isso o analista pode recalibrar valores de filtros e obter uma nova visualização dos rastros sumarizados em uma *Tree View*. Em sistemas complexos os diagramas podem ser difíceis de compreender, mesmo após todo o processo de sumarização dos rastros. Através da execução do algoritmo de agrupamento de subárvores é possível construir diagramas de sequência com níveis de detalhes diferentes. Com o objetivo de avaliar a abordagem são apresentados o processo de extração de visões de alto nível de uma árvore simplificada, onde o analista pode inserir valores de filtros para simplificar a análise visual.

Capítulo 5 - Trabalhos Correlatos. Este capítulo descreve várias técnicas de análise de rastros de execução, incluindo abordagens de sumarização e compressão de rastros como também técnicas para compreensão de sistemas de *software*.

Capítulo 6 - Conclusão e Trabalhos Futuros. Este capítulo apresenta as contribuições deste trabalho como também perspectivas para trabalhos futuros.

Capítulo 2

Coleta e Sumarização de Rastros de Execução

Neste capítulo é apresentado o processo de coleta e sumarização de rastros de execução em um sistema. Para que os rastros sejam capturados, o sistema alvo precisa ser instrumentado e uma determinada característica ser executada. Assim, todas as chamadas de métodos usadas durante a execução desta característica são armazenadas. O objetivo de sumarizar rastros extremamente volumosos é extrair uma visão de um rastro que seja entendível por analistas ou desenvolvedores na tentativa de compreender o comportamento de uma característica específica de um sistema. O algoritmo de sumarização reduz os rastros por remover chamadas de métodos que representam detalhes de implementação. A Figura 2.1 mostra uma visão geral dos principais passos da abordagem proposta de sumarização de rastros volumosos. Primeiro, o sistema alvo deve ser recompilado com suporte a aspectos para a coleta do rastro pela ferramenta de instrumentação, baseada em AspectJ. Em seguida os cenários de execução são executados, resultando em arquivos com a sequência de chamada de métodos. Após realizado a coleta dos rastros inicia o processo de compressão e sumarização. No primeiro passo os rastros são comprimidos para remover laços e chamadas recursivas. Em seguida é executado sobre os rastros comprimidos o algoritmo de sumarização, cujo objetivo é reduzir a complexidade dos rastros auxiliando na extração de informação importante. O algoritmo de compressão é executado antes e depois da sumarização para remover repetições que não foram encontradas na primeira execução. Após os rastros serem sumarizados, estes então são separados em novos arquivos onde cada arquivo contém somente chamadas de métodos executadas para cada tarefa contida no cenário.

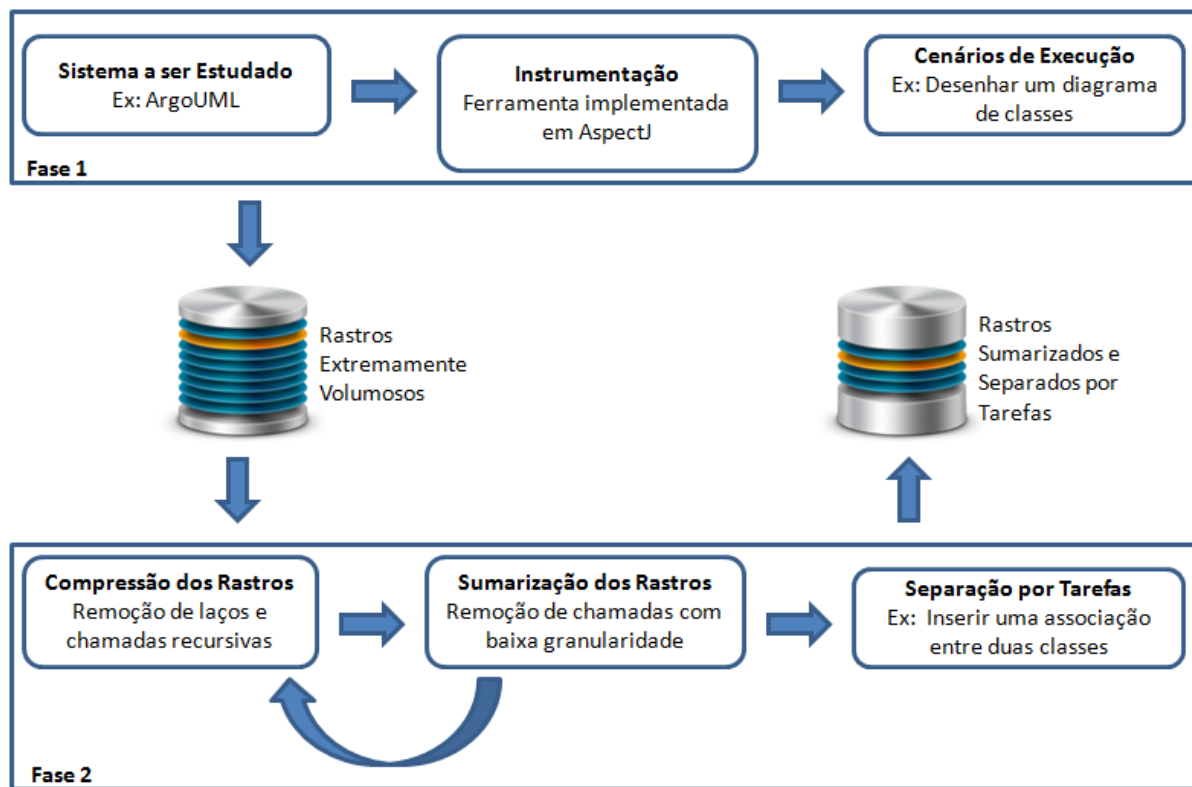


Figura 2.1: Principais passos do processo de sumarização de rastros.

2.1 Instrumentação do Sistema e Execução dos Cenários de Execução

O sistema alvo é instrumentado com a ferramenta de instrumentação interna baseada em AspectJ que captura os métodos executados durante a execução de um cenário pré-definido. A ferramenta pergunta ao desenvolvedor/analista para informar o início de uma tarefa do cenário. Quando a execução desta tarefa termina, tal evento deve ser informado. Este processo se repete até que todas as tarefas do cenário tenham sido executadas. Durante a execução de cada cenário, um arquivo de rastro é gerado para cada *thread* disparada. Cada linha do arquivo de rastro corresponde a uma chamada de método, que tem o nome qualificado, o nível da chamada na pilha de execução e o seu *timestamp*. Esta informação permite produzir arquivos de rastros sumarizados separados por cada tarefa do cenário. Por exemplo, a Tabela 2.1 mostra um exemplo de cenário de execução para uma ferramenta de modelagem UML ArgoUML¹. A execução deste cenário resultou em 6 arquivos de rastros, um para cada *thread* disparada. Após a separação por tarefa, dois arquivos de rastros foram criados para o passo “2. Criar Diagrama de Classes”, um que foi obtido do rastro da *Thread 2* e o outro da *Thread 6*. Estes dois arquivos de rastros contêm somente chamadas de métodos executadas para criar o diagrama de classes.

¹<http://argouml.tigris.org/>

Tabela 2.1: Par de Cenários para o ArgoUML

A. Desenhar Diagrama de Classes
1. Inicializar o Sistema
2. Criar Diagrama de Classes
3. Adicionar Classe C1
4. Adicionar Classe C2
5. Inserir uma Associação entre C1 e C2

Obviamente, as *threads* 2 e 6 podem ter sido usadas para executar outras tarefas, com isso a separação por tarefa seleciona somente as chamadas de métodos destas duas *threads* que foram executadas somente para o passo “2. Criar Diagrama de Classes”.

2.2 Algoritmo de Sumarização dos Rastros

O objetivo deste algoritmo é sumarizar os rastros de execução que em muitas vezes são muito volumosos. Mesmo após a execução do algoritmo de compressão proposto por [Hamou-lhadj e Lethbridge 2002] para remover chamadas recursivas e laços, extrair informação relevante pode ser uma tarefa muito difícil porque ruídos como eventos de *mouse* não são detectados pelo algoritmo e algumas repetições ainda permanecem no rastro comprimido. A Figura 2.2 mostra o algoritmo de compressão de rastros.

A técnica de sumarização de rastros proposta neste trabalho é baseada na eliminação

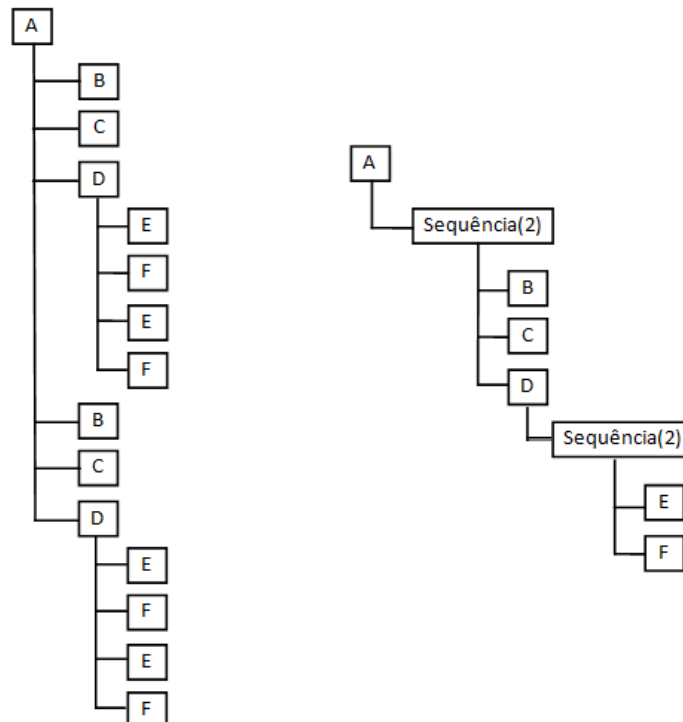


Figura 2.2: Exemplo de compressão dos rastros de execução.

Entrada: O rastro de execução comprimido na **Fase 1** $compressed_{trace}$ e a granularidade mínima $size_{minimum}$

Saída: O rastro sumarizado e recomprimido, contendo somente chamadas de métodos com granularidade mínima de $size_{minimum}$ e com menos repetições

1. **Encontra_Granularidade**($compressed_{trace}$)
2. **Sumariza_Rastro**($compressed_{trace}, size_{minimum}$)
3. **Aplique** o compressor novamente sobre o rastro sumarizado
4. **Salve** o rastro sumarizado-comprimido em arquivo

Figura 2.3: Algoritmo *Summarizer* com o parâmetro da granularidade mínima.

Entrada: O rastro de execução comprimido na **Fase 1** $compressed_{trace}$

Saída: A granularidade de cada chamada de método do rastro comprimido

1. **Para todas chamadas de métodos** $M_i \in compressed_{trace}$
2. **Encontre** a granularidade de M_i
3. **Salve** a granularidade de M_i

Figura 2.4: Algoritmo do método **Encontra_Granularidade**.

Entrada: O rastro de execução comprimido na **Fase 1** $compressed_{trace}$ e a granularidade mínima $size_{minimum}$

Saída: O rastro sumarizado contendo somente chamadas de métodos com granularidade mínima de $size_{minimum}$

1. **Para todas chamadas de métodos** $M_i \in compressed_{trace}$
2. $calls_{internal} := \text{Recupere}$ a granularidade de M_i
3. **Se** $calls_{internal} \geq size_{minimum}$ **Então**
4. **Selecione** M_i
5. **Salve** o rastro sumarizado em arquivo

Figura 2.5: Algoritmo **Sumariza_Rastro** com o parâmetro da granularidade mínima.

de chamadas de métodos com baixa granularidade. A granularidade de uma chamada é calculada sobre o número de chamadas que ocorreram dentro da chamada atual. Quanto maior for o número de chamadas internas, maior é a granularidade e a importância relativa sobre as outras chamadas. Assume-se como hipótese que, chamadas de métodos com granularidades maiores em um rastro são os métodos mais relevantes para a execução de uma determinada característica. Quanto menor é a granularidade mínima escolhida, menor é o número de chamadas de métodos removidas do rastro e assim maior é o número de possíveis informações irrelevantes para analisar, porém menor é a chance de descartar chamadas de métodos relevantes. Para encontrar a granularidade mínima mais adequada, o analista começa com a granularidade igual a 1 e analisa o resultado correspondente. Se ainda contém muita informação para analisar, a granularidade pode ser aumentada. Se a granularidade maior elimina chamadas de métodos relevantes, então a granularidade anterior é considerada a ideal. A identificação de chamadas relevantes/irrelevantes depende da percepção do analista, sendo portanto subjetiva. As Figuras 2.3, 2.4 e 2.5 mostra como é feito a sumarização dos rastros com o parâmetro de granularidade mínima.

Com o intuito de descobrir qual seria a granularidade mínima adequada para a su-

Entrada: O rastro de execução comprimido na **Fase 1** $summarized_{trace}$ e o tamanho do rastro original N_{raw}

Saída: O rastro sumarizado com no mínimo 90% de compactação em relação ao rastro original

1. $N_{current} := N_{raw}$
2. **Enquanto** $|\frac{N_{current}}{N_{raw}} - 1| < 0.9$ **faça**
3. **Execute** o **Summarizer** sobre $summarized_{trace}$ com granularidade mínima igual a 1
4. $N_{current} :=$ tamanho atual de $summarized_{trace}$

Figura 2.6: Algoritmo *Summarizer* automatizado.

marização dos rastros sem perda de informações relevantes, foram feitos vários testes preliminares sobre diferentes cenários em sistemas distintos. Após as análises, pôde-se observar que boas sumarizações ocorreram com taxa de sumarização em torno 93%. Com base nestes resultados o algoritmo de sumarização apresentado na Figura 2.6 foi automatizado, sem a necessidade do analista informar a granularidade mínima.

2.2.1 Análise do Algoritmo

O algoritmo *Summarizer* apresentado na Figura 2.3 contém duas rotinas. A análise deste algoritmo inicia na primeira rotina que está apresentada na Figura 2.4.

Análise da rotina *Encontra_Granularidade*. Para estimar o tempo de execução considere n o número de chamadas no rastro comprimido. O passo 1 é executado n vezes, $O(n)$. O passo 2 possui tempo linear no pior caso, $O(n)$, quando o rastro contém somente uma única sequência de chamadas de métodos que é uma situação pouco provável de acontecer em um rastro de execução. No melhor caso este passo possui tempo de execução constante $O(1)$ que significa que o rastro possui várias sequências de chamadas de métodos. O passo 3 possui tempo de execução constante $O(1)$. Logo, o tempo de execução do algoritmo no pior caso é de $O(n)*O(n)*O(1) = \Theta(n^2)$ e no melhor caso o tempo é linear $O(n)*O(1)*O(1) = \Theta(n)$. O espaço usado é baseado no tamanho do rastro comprimido, ou seja, n chamadas de métodos, onde cada linha do rastro contém informações de cada chamada. Logo, o espaço necessário é $\Theta(n)$, porque esta rotina guarda somente a granularidade de cada chamada de método na sua respectiva linha do rastro.

A segunda rotina do algoritmo *Summarizer*, apresentada na Figura 2.5, remove todas as chamadas de métodos com granularidade menor que a granularidade mínima.

Análise da rotina *Sumariza_Rastro*. Para estimar o tempo de execução considere n o número de chamadas no rastro comprimido. O passo 1 é executado n vezes, $O(n)$. Cada execução dos passos 2, 3, 4 possui tempo de execução constante, $O(1)$. O passo 5 também possui tempo constante, $O(1)$. Logo, o tempo de execução desta rotina é linear $O(n)*O(1)*O(1)*O(1)+O(1) = \Theta(n)$. O espaço usado para o rastro sumarizado é menor

que o espaço usado para o rastro comprimido, dependendo muito de duas situações, do valor de parâmetro *granularidade mínima* e a granularidade das chamadas do rastro. Seja m o número de chamadas de métodos que foram removidas do rastro comprimido. Então, $0 < m < n$ por causa das duas situações apresentadas. Logo, o espaço necessário é no máximo, $\Theta(n)$.

A partir da análise das rotinas que constituem o algoritmo *Summarizer*, Figura 2.3, é possível obter o seu tempo total de execução e o espaço necessário para a sua execução. O passo 1 contém $\Theta(n^2)$ no pior caso. O passo 2 contém tempo linear, $\Theta(n)$. O passo 3 possui tempo $O(n)$, a análise foi apresentada em [Hamou-lhadj e Lethbridge 2002]. O passo 4 possui tempo constante $O(1)$. Logo, o tempo total é $\Theta(n^2) + \Theta(n) + O(n) + O(1) = \Theta(n^2)$ no pior caso e $\Theta(n) + \Theta(n) + O(n) + O(1) = \Theta(n)$ no melhor caso. Por fim, o espaço total usado pelo algoritmo é $\Theta(n) + \Theta(n) + \Theta(n) + \Theta(n) = \Theta(n)$.

Para estimar o tempo de execução do algoritmo *Summarizer* automatizado, apresentado na Figura 2.6, considere n o número de chamadas no rastro comprimido. Considere também o número de vezes que o passo 2 é executado. Suponha k este número, onde k é sempre um número muito pequeno em relação n . Portanto, k pode ser considerado como constante tal que $O(k) = O(1)$. O passo 1 é executado somente uma vez e possui tempo constante, $O(1)$. Cada execução do passo 3 possui tempo $\Theta(n^2)$ no pior caso e $\Theta(n)$ no melhor caso. Cada execução do passo 4 possui tempo constante, $O(1)$. Logo, o tempo total de execução do algoritmo é $O(1) * \Theta(n^2) * O(1) = \Theta(n^2)$ para o pior caso e $O(1) * \Theta(n) * O(1) = \Theta(n)$ para o melhor caso. O espaço total usado pelo algoritmo é o mesmo que o apresentado da Figura 2.6.

2.2.2 Representação dos Rastros em uma *Tree View*

Os rastros de execução podem ser representados em uma *Tree View*, facilitando a extração de informações relevantes para a compreensão de uma determinada tarefa do sistema alvo. A Figura 2.7 mostra um fragmento de rastro de execução sumarizado de

```
org.argouml.application.Main.initializeSubsystems,2,-1,1287494761169,[]
org.argouml.application.Main.initModel,3,-1,1287494761169,[null]
org.argouml.model.Model.initialise,4,-1,1287494762484,[org.argouml.model.mdr.MDRModelImplementation
org.argouml.model.mdr.MDRModelImplementation.getDefaultRepository,5,-1,1287494763012,[null]
org.argouml.model.mdr.MDRModelImplementation.<init>,5,51,1287494764497,[null]
org.argouml.model.mdr.MDRModelImplementation.createDefaultExtent,6,51,1287494766892,[null]
org.argouml.model.mdr.MDRModelImplementation.initializeFactories,6,51,1287494767362,[]
org.argouml.model.mdr.ModelEventPumpMDRImp.<init>,7,53,1287494767462,[null]
org.argouml.model.mdr.ModelEventPumpMDRImp.buildTypeMap,8,53,1287494767520,[]
org.argouml.model.mdr.ModelEventPumpMDRImp.getSubtypes,10,53,1287494767521,[]
org.argouml.model.mdr.ModelEventPumpMDRImp.getSubtypes,10,53,1287494767548,[]
org.argouml.model.mdr.ModelEventPumpMDRImp.buildPropertyNameMap,8,53,1287494767571,[]
org.argouml.model.Model.setImplementation,5,-1,1287494769034,[]
org.argouml.application.Main.updateProgress,3,-1,1287494769138,[5
org.argouml.profile.internal.InitProfileSubsystem.init,3,77,1287494769676,[null]
org.argouml.profile.internal.ProfileManagerImpl.<init>,4,80,1287494770367,[null]
org.argouml.profile.internal.ProfileUML.<init>,5,81,1287494770481,[null]
org.argouml.profile.ResourceModelLoader.loadModel,6,83,1287494770738,[]
```

Figura 2.7: Fragmento de rastro de execução sumarizado da inicialização do ArgoUML.

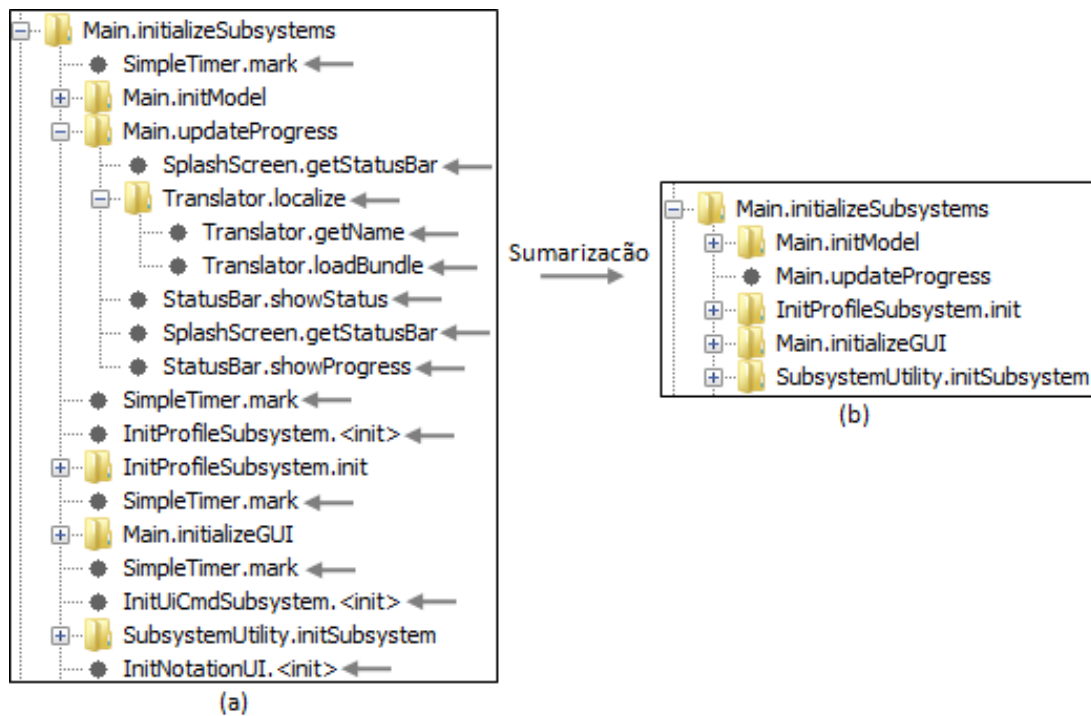


Figura 2.8: (a) mostra um rastro comprimido, onde os laços e recursões foram removidos e (b) mostra o rastro após a execução do algoritmo *Summarizer*.

inicialização do sistema de modelagem ArgoUML. Existem seis colunas nos rastros, a primeira coluna é o nome do pacote e da classe, a segunda é o nome do método, a terceira é a ordem da chamada de método na pilha de execução, a quarta o id do objeto e a quinta contém os conteúdos de parâmetros dos métodos. A terceira coluna é utilizada para gerar a *Tree View*. A Figura 2.8.a mostra o fragmento de rastro do ArgoUML comprimido representado em uma *Tree View*. As setas cinzas mostram as chamadas que serão podadas durante a execução do algoritmo de *Summarizer* para a granularidade mínima igual 3. A Figura 2.8.b mostra o rastro sumarizado da Figura 2.7 em uma *Tree View*. Pode-se observar que existem alguns nós na árvore que podem ser expandidos, apresentando chamadas de métodos que estão ocultas. Um exemplo é a terceira linha do rastro mostrado na Figura 2.7 que para visualizá-la basta expandir o nó `Main.initModel` da *Tree View*.

2.3 Separação por Tarefas

Durante a coleta dos rastros de execução, um arquivo de rastro é criado para cada *thread* disparada. A ferramenta de instrumentação, apresentada na Seção 2.1, marca o tempo de início e fim de execução de cada tarefa de um cenário de execução. A partir desta informação é possível separar as chamadas de métodos que foram executadas para uma determinada tarefa do cenário, como também detectar a importância de cada *thread*

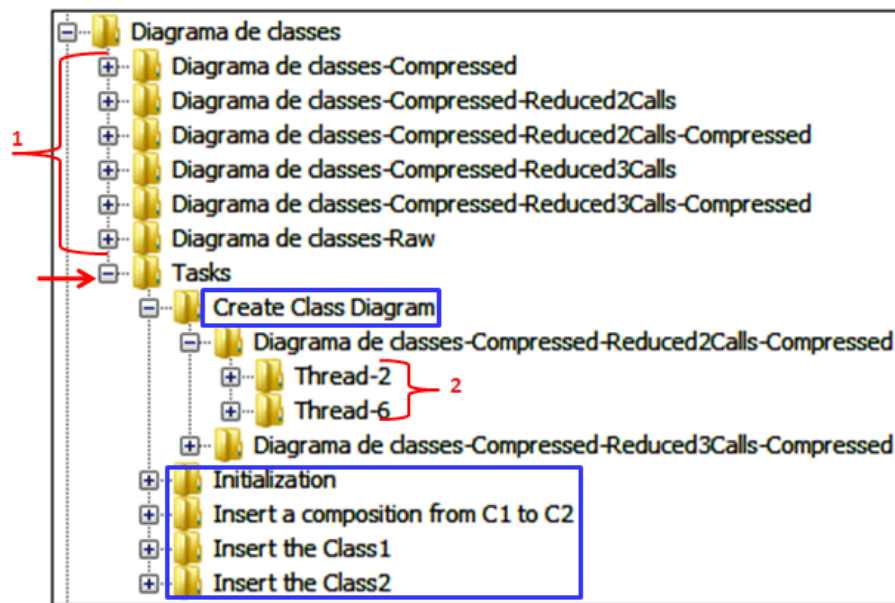


Figura 2.9: Exemplo de repositório de rastros após a execução da separação por tarefas.

para uma característica.

A Figura 2.9 mostra um repositório do sistema ArgoUML, onde contém os rastros sumarizados e comprimidos que estão marcados pela chave 1. A seta vermelha marca a pasta que contém todas as tarefas do cenário executado e suas respectivas tarefas, delineadas em azul. A chave 2 mostra que as *Threads* 2 e 6 são responsáveis pela criação da tarefa “Criar Diagrama de Classes” e nestes arquivos de *threads* contém somente chamadas de métodos usadas para criar o diagrama de classes. Desta maneira, os arquivos de rastros sumarizados ficam menores, contendo somente as chamadas de métodos da tarefa a ser analisada com o propósito de reduzir o esforço de compreensão.

2.4 Considerações Finais

Neste capítulo foi apresentado o processo de sumarização dos rastros de execução. Os passos contidos neste processo são passos em comum na aplicação das técnicas apresentadas nos Capítulos 3 e 4 deste trabalho. O primeiro passo descreve que o sistema alvo deve ser instrumentado e em seguida um cenário pré-definido é executado. No segundo passo, os rastros capturados durante a execução do cenário são comprimidos, para remover as repetições, e em seguida é executado o algoritmo de sumarização. O algoritmo *Summarizer* é detalhado neste capítulo, mostrando os seus pseudocódigos e ilustrações de como um rastro de execução é representado em uma *Tree View*.

Capítulo 3

Alinhamento de Rastros de Execução Sumarizados

O objetivo da aplicação da técnica de alinhamento de rastros de execução sobre rastros sumarizados é separar chamadas de métodos que são específicas a uma característica das chamadas que são comuns para direcionar a compreensão de sistemas e reduzir o esforço de implementar novas características no sistema alvo.

A aplicação desta técnica planeja contribuir com o seguinte cenário. Supondo a existência de um sistema de *software* com código disponível. Supondo a pré-condição que os desenvolvedores responsáveis por introduzir uma nova característica neste *software* não tenham participado previamente do processo de desenvolvimento e que não tenham conhecimentos prévios do sistema. Supondo também que não exista documentação ou os desenvolvedores não querem usar a documentação existente porque não é confiável. Por exemplo, supondo que os desenvolvedores querem evoluir uma ferramenta de modelagem UML com mais de cem mil linhas de código para inserir um novo tipo de diagrama. Este cenário impõe um desafio maior para a equipe. Não se pode esperar que esta tarefa seja realizada com um tempo determinado.

3.1 Algoritmo de Alinhamento de Rastros de Execução

Nesta seção algumas definições para o problema de alinhamento de sequências são apresentadas. Uma contribuição importante para o problema de alinhamento de sequências foi proposta por Needleman e Wunsch [Needleman e Wunsch 1970]. Eles sugeriram um algoritmo de programação dinâmica para executar alinhamento global sobre duas sequências. Este algoritmo foi adaptado para propósitos de alinhamento de rastros de execução por [Paixão 2009]. A primeira adaptação é sobre a característica de um rastro de execução. Os rastros podem conter ruídos. Por exemplo, nas sequências “XaaaaY” e “XaaaY”, pode supor que estas sequências foram construídas pelo mesmo código fonte. A

diferença poderia ter sido causada por um laço que executou o método “a” quatro vezes na primeira sequência e três na segunda e que é por isso que os rastros são comprimidos.

Existem dois tipos de alinhamento: Alinhamento Local e Global [Gusfield 1997]. O objetivo do alinhamento global é comparar sequências com alto nível de similaridade e tamanhos similares. O alinhamento local é mais adequado para situações onde o foco é encontrar uma sequência menor dentro de uma sequência maior. O alinhamento global foi o escolhido porque é usado para comparar sequências com um nível alto de similaridade com tamanhos similares. O objetivo é encontrar características similares e pontos de variação nas sequências de rastros.

Uma segunda possibilidade para algoritmos de alinhamento é o número de sequências que podem ser alinhadas simultaneamente. O alinhamento de um par de sequências é usado no alinhamento global e local. O alinhamento de múltiplas sequências é indicado para sequências pequenas porque este problema é NP-completo. Como as sequências de rastros a serem comparadas são muito grandes, o alinhamento de par foi a abordagem viável.

Uma terceira possibilidade para algoritmos de alinhamento é sobre a decisão, se as sequências irão ser alinhadas considerando regras de substituição entre os elementos. Neste primeiro caso, uma matriz de substituição deve ser definida e o *score* é calculado considerando os valores atuais na matriz. Regras de substituição são indicadas quando existe uma relação de similaridade entre os elementos e elementos diferentes podem alinhar dependendo em suas similaridades. O alinhamento de rastros de execução pode ser feito somente entre elementos idênticos porque não parece ter utilidade a relação de similaridade entre nomes de métodos diferentes pelos propósitos do trabalho em [Paixão 2009].

Dois rastros de alinhamento podem ser analisados em segmentos. Segmentos de rastros são agrupados de elementos alinhados consecutivamente e também de grupos de elementos desalinhados consecutivamente. A Figura 3.1 mostra um exemplo de alinhamento de

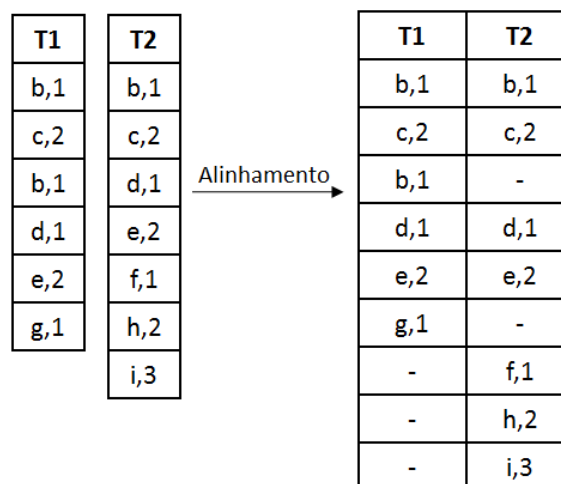


Figura 3.1: Exemplo de alinhamento entre duas sequências de rastros de execução.

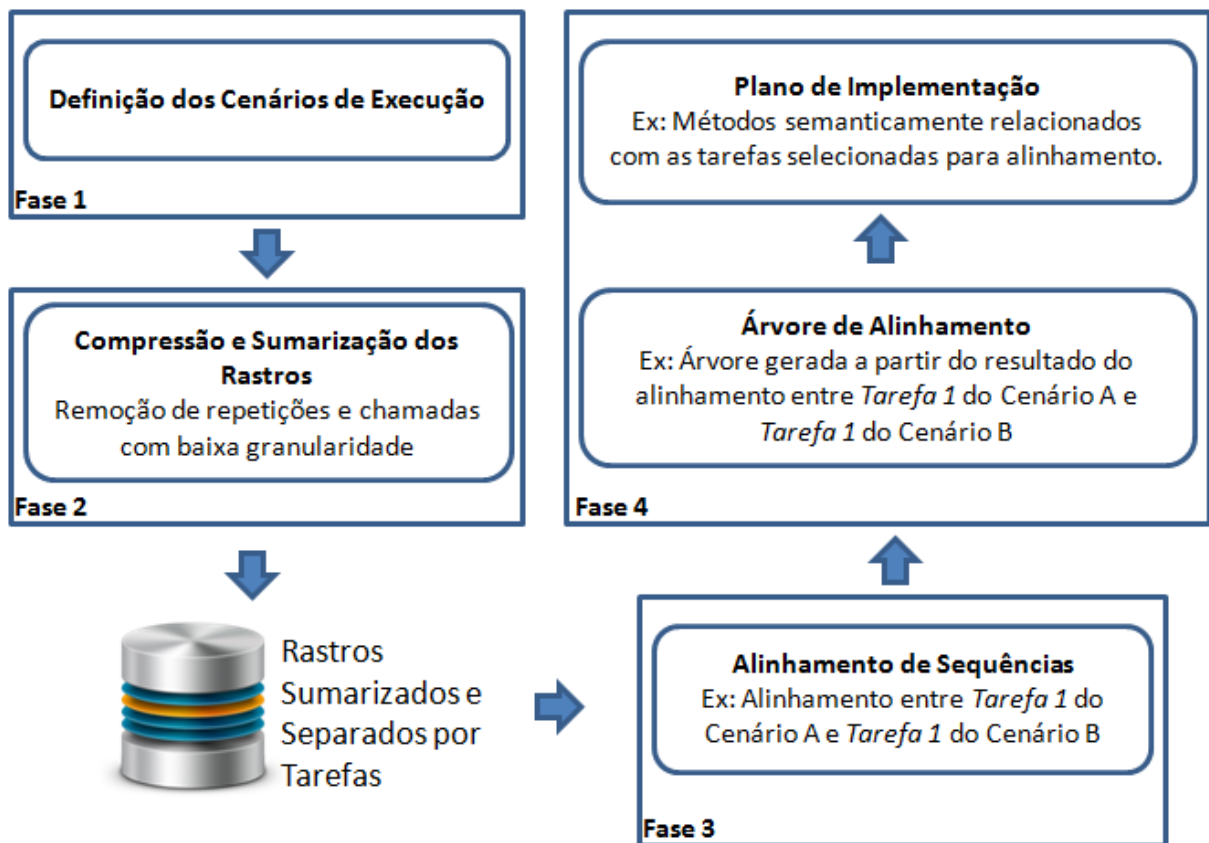


Figura 3.2: Descrição das 4 fases da técnica de alinhamento.

duas sequências de rastros. A primeira e a segunda linha dos dois rastros se alinham. A terceira linha $(b, 1)$ não se alinha com $(d, 1)$, então é inserido um espaço na mesma posição de $(b, 1)$ no rastro T2. Em seguida a quarta linha do rastro T1 se alinha com $(d, 1)$ no rastro T2. Esta comparação é feita em todas as linhas dos dois rastros.

3.2 Processo de Alinhamento de Rastros Sumarizados

A Figura 3.2 mostra uma visão geral da abordagem proposta que possui quatro fases principais:

- Definição dos Cenários de Execução
- Sumarização dos Rastros de Execução
- Alinhamento dos Rastros Sumarizados
- Definição do Plano de Implementação

3.2.1 Definição dos Cenários de Execução

Na primeira fase, a definição dos cenários é baseada na busca por características semelhantes no sistema que são similares com a nova a ser inserida, mas que tenham alguma

Tabela 3.1: Exemplo de cenários para detectar código comum e específico que implementam as características.

Cenário A	Cenário B
1. Inicializar o Sistema	1. Inicializar o Sistema
2. Desenhar Retângulo	2. Desenhar Círculo
3. Copiar Retângulo	3. Copiar Círculo

especificidade entre elas. A informação obtida desta especificidade deve guiar a implementação do plano.

Escolha de Características Adequadas em Pares

Com o interesse de reusar, modificar ou inserir novas características em um sistema, primeiro o desenvolvedor precisa saber quais são as características básicas que o sistema executa. Por exemplo, suponha que o sistema é um cliente de *email*. O desenvolvedor precisa saber que *enviar mensagem*, *receber mensagem*, *salvar como rascunho* e *efetuar login* são características básicas do sistema. O desenvolvedor pode definir quais características estão relacionadas uma com a outra. Considere o seguinte exemplo de um simples editor de figura. Suponha que as características *Desenhar Retângulo* e *Desenhar Círculo* já estão implementadas no sistema e o desenvolvedor precisa implementar a característica *Desenhar Triângulo*. A Tabela 3.1 mostra um exemplo de um par de cenários de execução A e B para detectar código comum e específico que implementa as características *Desenhar Retângulo* e *Desenhar Círculo*. Algumas chamadas de métodos, aquelas usadas para criar os itens de menu, podem estar relacionadas com a tarefa “1. Inicialização do sistema”. Também, durante a execução de *Desenhar Retângulo* e *Desenhar Círculo*, talvez exista algumas chamadas de métodos que são comuns em ambas tarefas. Chamadas de métodos específicas para desenhar cada tipo de figura podem estar relacionadas com as respectivas tarefas nos cenários A e B. A Tabela 3.1 também mostra as tarefas, *Copiar Retângulo* e *Copiar Círculo*, que talvez ajude a compreender como introduzir *Copiar Triângulo* no sistema.

No entanto, é possível tentar comparar os dados de rastros que contêm a execução de uma característica F1 de interesse com os dados de rastros que contêm a execução de uma característica F2 totalmente diferente usada somente para controlar. Uma decisão importante na definição dos cenários de execução é que o desenvolvedor introduz tarefas comuns na mesma ordem em ambos cenários, possivelmente que envolve a execução de F1 e F2 para controlar onde aqueles rastros têm métodos relacionados a F1 e F2, respectivamente, que podem ser capturados pelo algoritmo de alinhamento.

A fase de sumarização dos rastros foi apresentada no Capítulo 2 porque esta fase é idêntica no Capítulo 4 deste trabalho.

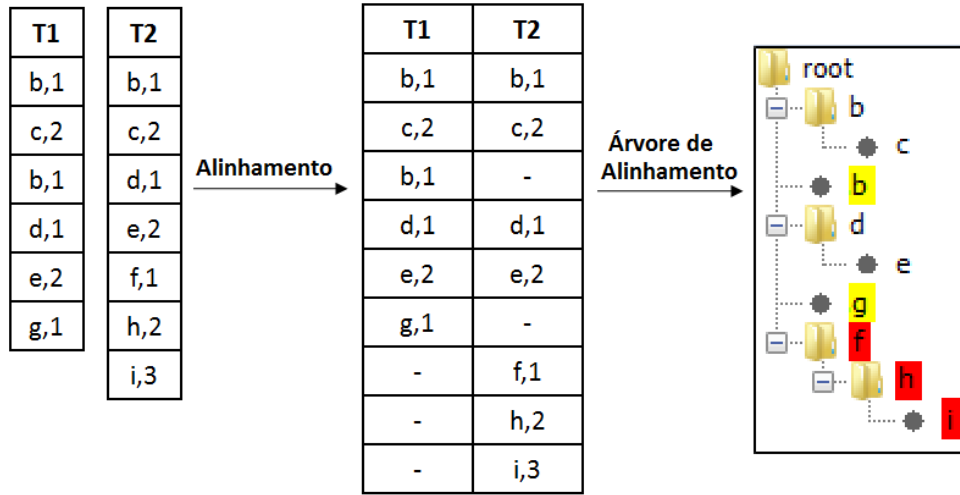


Figura 3.3: A abordagem de alinhamento de rastros sumarizados.

3.2.2 Alinhamento dos Rastros de Execução Sumarizados

A técnica de alinhamento de rastros é aplicada sobre os rastros sumarizados para viabilizar o uso da abordagem proposta por [Paixão 2009]. A entrada da ferramenta de alinhamento são duas sequências de rastros sumarizados de tamanhos diferentes possivelmente. No final do alinhamento, duas novas sequências com o mesmo tamanho são construídas, contendo espaços inseridos tal que os métodos alinhados estão na mesma posição em ambas sequências.

Uma hipótese do alinhamento de rastros é que pontos desalinhados nas sequências indicam chamadas de métodos de características diferentes. Portanto, pontos desalinhados sugerem métodos que deveriam ser analisados para completar a implementação da nova característica. No entanto, os desalinhamentos também podem representar ruídos no rastro como eventos de mouse que não foram removidos pelo algoritmo *Summarizer*.

Além disso, como resultado do alinhamento é também possível identificar sequências de métodos alinhados em ambas sequências que foram executados antes e depois dos métodos desalinhados. Então, o plano de implementação é projetado com a informação de que métodos considerados relevantes ajudarão possivelmente na implementação da nova característica.

Neste trabalho, o resultado do alinhamento de um par de rastros é representado em uma *Tree View* porque mostra as chamadas de métodos aninhadas. Os segmentos desalinhados são mostrados em diferentes cores onde nós com a mesma cor são da mesma sequência. A Figura 3.3 ilustra um exemplo de alinhamento de rastro. Dois rastros são apresentados contendo as chamadas de métodos (b, c, d, e, f, g, h, i) e seus respectivos níveis na pilha de execução. A sequência $\langle (b, 1), (c, 2) \rangle$ indica que o método *c* foi chamado do método *b*. Na árvore de alinhamento, o primeiro segmento apresenta as chamadas *b* e *c*, onde a chamada *c* é filha de *b*. O segundo segmento contém um desalinhamento indicando

que a chamada para o método b no rastro T1 não tem uma contraparte no rastro T2. O próximo segmento alinhado apresenta as chamadas d e e , onde a chamada de método e é um filho da chamada d . Após este segmento existe um segmento desalinhado, contendo somente as chamadas de método g no rastro T1, mas não possui a contraparte no rastro T2. Finalmente, o último segmento desalinhado indica que a sequência de chamadas de métodos f , h e i no rastro T2 não tem a contraparte no rastro T1 na mesma posição.

3.2.3 Definição do Plano de Implementação

Este passo é uma análise visual das árvores de alinhamento obtidas na fase anterior. Algumas *threads* e alguns blocos desalinhados cujos nomes de métodos, classes e pacotes não estão relacionados semanticamente com o cenário de execução podem ser descartados. Durante a execução do cenário, o desenvolvedor informou o início de cada passo do cenário. Esta informação permitiu produzir os arquivos de rastros sumarizados separados para cada tarefa do cenário. Se o cenário de execução disparar mais de uma *thread* então é possível saber quais *threads* executaram uma tarefa. Portanto, para entender como uma tarefa está implementada, o desenvolvedor seleciona algumas *threads* que contém chamadas de métodos executadas para a tarefa que está sendo analisada.

O projeto do plano de implementação então inicia por inserir métodos que ajudariam a compreender como uma característica está implementada no código fonte. Métodos relevantes para executar uma tarefa são selecionados das árvores de alinhamento para guiar na implementação de uma nova característica. Os métodos são selecionados como importantes por uma simples inspeção dos nomes das classes, métodos e pacotes. Após analisar as árvores de alinhamento, os métodos selecionados são adicionados no plano de implementação.

Avaliação de Qualidade do Plano de Implementação

A qualidade da lista de métodos para o plano de implementação deve ser avaliada. Estes métodos foram listados pela abordagem proposta neste trabalho como sendo importantes para conduzir as mudanças perfectivas. Neste estudo foi definido o controle para avaliar a qualidade dos métodos no plano de implementação. Os métodos **Reais Positivos** (*RP*) são os métodos usados durante a implementação atual da nova característica.

Os métodos incluídos no conjunto RP são aqueles que foram necessários para compreender a mudança necessária e aqueles que foram efetivamente modificados. Através dos métodos no conjunto RP e o plano de implementação é possível descobrir os valores dos seguintes termos:

VP: Verdadeiros Positivos. Métodos que foram apontados pela abordagem e estes foram efetivamente usados.

FP: Falsos Positivos. Métodos apontados pela abordagem e que não foram usados.

FN: Falsos Negativos. Métodos que foram necessários mas não foram apontados pela abordagem.

Estes valores são utilizados para avaliar a abordagem em termos de precisão¹ e *recall*².

3.3 Estudos de Caso e Avaliação dos Resultados

O objetivo da avaliação é determinar como a abordagem melhora a prática do desenvolvedor quando executa mudanças perfectivas no sistema de *software*. Na verdade, uma medida razoável que avaliaria a melhora hipotética é o esforço necessário para executar a tarefa inteira de compreender o sistema alvo e implementar as mudanças requeridas. Esta opinião requer um estudo experimental com um número razoável de desenvolvedores executando a tarefa, usando uma abordagem tradicional e usando esta abordagem. Ao invés, como um primeiro estudo, neste trabalho o intuito é entender como a abordagem proposta ajudaria o desenvolvedor em fornecer potencialmente informação precisa e suficiente para guiar na execução de tarefas de modificações. Esta escolha leva a um estudo que analisa a precisão e *recall* da informação fornecida pela abordagem de alinhamento. Três estudos são apresentados com sistemas de *software* diferentes que utiliza a abordagem de alinhamento de rastros de execução sumarizados para direcionar a evolução de sistemas selecionados.

Os sistemas selecionados são *software* de código aberto disponíveis em um repositório público em domínios de aplicações diferentes. As mudanças perfectivas foram selecionadas para ser significantes e não somente mudanças de apresentação, como mudar uma propriedade da GUI.

Os estudos incluem várias fases para definir um conjunto de métodos sugeridos que os desenvolvedores deveriam considerar para efetuar a mudança perfectiva. A avaliação deste conjunto de métodos requer efetivamente implementar as mudanças e armazenar os métodos que foram usados de verdade na implementação e então o conjunto de resultados de métodos pode ser avaliado em termos de precisão e *recall*.

3.3.1 Estudo de Caso 1 - ArgoUML

O ArgoUML³ 0.30 é uma ferramenta de modelagem UML com aproximadamente 161 KLOC, com 92 pacotes, 2.122 classes, 156 interfaces, 2.314 atributos, 1.914 atributos estáticos, 13.918 métodos e 669 métodos estáticos. As bibliotecas como, o *framework* GEF - *Graph Editing Framework* e o metamodelo da UML que o ArgoUML utiliza não

¹É a proporção das chamadas de métodos relevantes que foram selecionados pela abordagem, ex.: $VP/(VP + FP)$

²É a proporção das chamadas de métodos selecionadas que a abordagem selecionou corretamente, ex.: $VP/(VP + FN)$

³[http : //argouml.tigris.org/](http://argouml.tigris.org/)

estão incluídos. A mudança perfectiva escolhida para este sistema é a adição de um novo diagrama: Diagrama de Objetos.

As três fases seguintes foram executadas para obter um conjunto de métodos sugeridos para planejar a implementação das modificações.

Definição e Execução dos Cenários

Os cenários foram especificados buscando por diagramas existentes que eram similares para o novo diagrama requerido. O diagrama de classes e o diagrama de componentes são similares porque eles são ambos baseados em nós e ligações entre eles. No entanto, o tipo de nós e ligações talvez sejam específicos para cada um. Os cenários selecionados, mostrados na Tabela 3.2 tem uma relação direta entre seus passos.

Tabela 3.2: Par de Cenários para o ArgoUML

A. Desenhar Diagrama de Classes	B. Desenhar Diagrama de Componentes
1. Inicializar o Sistema	1. Inicializar o Sistema
2. Criar Diagrama de Classes	2. Criar Diagrama de Componentes
3. Adicionar Classe C1	3. Adicionar Componente D1
4. Adicionar Classe C2	4. Adicionar Componente D2
5. Inserir uma Associação entre C1 e C2	5. Adicionar Dependência entre D1 e D2

Sumarização dos Rastros de Execução

Durante a execução de cada cenário, seis arquivos de rastros foram coletados, um para cada *thread* disparada. O compressor foi executado para remover os laços e chamadas recursivas. Então, o algoritmo *Summarizer* reduziu o tamanho dos arquivos comprimidos, eliminando chamadas de métodos com baixa granularidade. Métodos com granularidade 2 foram removidos. Uma tentativa de remover métodos com granularidade maior que ou igual a 3 foi descartada porque os rastros resultantes apresentaram uma influência maior nos resultados do alinhamento, podendo métodos específicos que eram claramente relacionados à característica estudada. Uma seleção adequada do parâmetro de granularidade será discutida na Subseção 3.4. Finalmente, o compressor foi aplicado novamente sobre os rastros sumarizados para remover possíveis laços que não foram detectados na primeira compressão. A Tabela 3.3 mostra o resultado de cada passo para reduzir os rastros dos cenários selecionados. A taxa de redução dos rastros está relacionada com o tamanho original. Durante a execução do cenário, o desenvolvedor deve informar o início e o fim de cada passo do cenário. Esta informação permite produzir arquivos de rastros sumarizados separados para cada passo do cenário. Por exemplo, 2 arquivos de rastros foram criados para o passo “*Criar Diagrama de Classes*”, um que foi obtido do rastro da *Thread 2* e o

Tabela 3.3: Resultados obtidos da execução dos algoritmos de Compressão e Sumarização dos Rastros de Execução do ArgoUML.

	Cenário A	Cenário B
Tarefas	5	5
Tamanho do Rastro Bruto	238.107	277.720
Tamanho do Rastro Comprimido	181.608	216.277
Taxa de Compressão	23,73%	22,12%
Tempo de Execução(<i>ms</i>)	26607	43087
Tamanho do Rastro Comprimido-Sumarizado	43.146	52.701
Taxa de Compressão	81,88%	81,02%
Tempo de Execução(<i>ms</i>)	4531	5375
Tamanho do Rastro Comprimido-Sumarizado-Comprimido	36.733	41.305
Taxa de Compressão	84,57%	85,13%
Tempo de Execução(<i>ms</i>)	3305	4157

outro do rastro da *Thread 6*. Estes 2 arquivos de rastros contêm algumas chamadas de métodos executadas para criar o diagrama de classes.

Alinhamento dos Rastros Sumarizados

Após a sumarização dos rastros, o algoritmo de alinhamento foi executado nos rastros sumarizados obtidos da fase anterior. Cada alinhamento foi obtido entre a *i*-ésima *thread* do cenário A e a *i*-ésima *thread* do cenário B. O resultado de cada alinhamento foi representado em uma árvore de alinhamento separada usada na próxima fase.

Definição do Plano de Implementação

O primeiro passo nesta fase é uma análise visual das árvores de alinhamento para definir os métodos selecionados que seriam relevantes à implementação da característica *Diagrama de Objetos*. Os métodos foram selecionados, a partir da árvore de alinhamento, por uma simples inspeção de nomes dos pacotes, classes e métodos. Antes da seleção

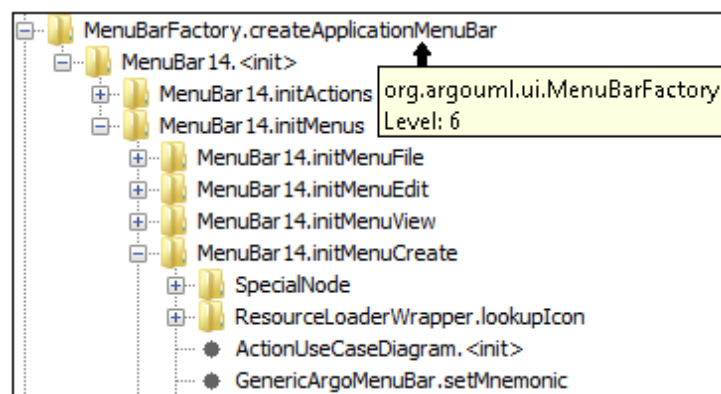


Figura 3.4: Fragmento da árvore que contém a inicialização do ArgoUml.

de métodos começar, algumas *threads* e alguns blocos desalinhados cujos métodos não estão semanticamente, próximos ao cenário de execução, são eleitos a serem descartados. As semânticas principais das *threads* poderiam ser analisadas por uma inspeção visual da árvore de alinhamento. Por exemplo, o alinhamento resultante da *Thread 2* entre o cenário A e B mostrou 73 blocos desalinhados. Após a análise destes blocos, 4 blocos desalinhados foram descartados. Por exemplo, `mouseExited` é um evento de *mouse* que não está semanticamente próximo com nenhuma ação importante do cenário de execução, então seu bloco correspondente foi descartado. Na verdade, existe um espaço para subjetividade neste critério e é possível que o desenvolvedor possa ter um critério de descarte agressivo, eliminando métodos importantes. Ao invés, nossa abordagem foi bastante conservativa porque somente 5% dos blocos desalinhados foram descartados. A *Thread 1* e a *Thread 2* tinham nomes de chamadas de métodos relacionadas diretamente com a inicialização do sistema como mostra a Figura 3.4. A *Thread 2* também tinha nomes de chamadas de métodos relacionadas com a criação do diagrama de classes, a adição de classes no diagrama e a adição de uma associação. A Figura 3.5 mostra um fragmento da árvore, ilustrando um desalinhamento ocorrido durante a criação dos diagramas de classes e componentes. O leitor atento talvez note que as chamadas de métodos quando é feito o desenho do diagrama de componentes são chamados da classe `ActionDeploymentDiagram`. A razão é que a paleta do diagrama de implantação foi usado para criar o diagrama de componentes. A *Thread 3* tinha chamadas de métodos de classes que desenhavam figuras de classes (`FigClass`) e as figuras de componentes (`FigMNode`). A semântica destas classes foi inferida por seus nomes, mas confirmada somente com a inspeção no código fonte. A conclusão foi que a *Thread 3* foi responsável por renderizar os gráficos. Na *Thread 2*

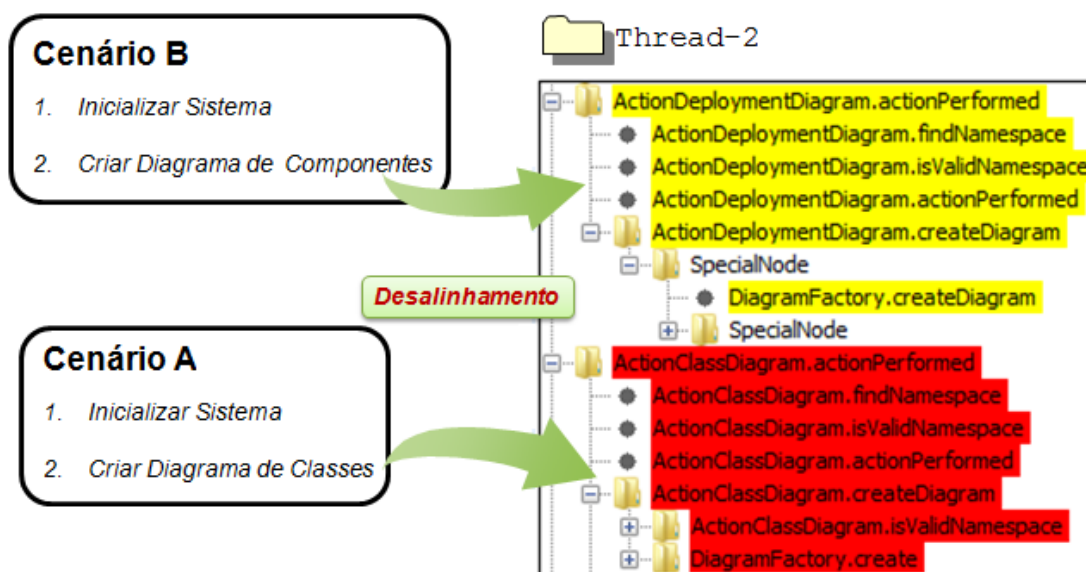


Figura 3.5: Fragmento da árvore que ilustra a sequência de chamada específica para cada cenário.

também tinha as classes `(FigClass)` e `(FigMNode)`, então a *Thread 3* foi descartada porque não forneceria outro tipo de informação relevante. Após estas análises, a hipótese de que uma classe chamada `(FigObject)` seria necessária. Uma busca por classes com o nome `(Fig)` revelou que esta classe já estava implementada e deveria ser considerada para reúso, se possível. Na *Thread 4* tinha chamadas de métodos relacionadas a críticas. Por questão de simplicidade, críticas não foram incluídas na implementação da característica Diagrama de Objetos, então esta *thread* poderia ser também descartada. A *Thread 5* foi descartada porque todas as chamadas de métodos foram alinhadas e elas também estão relacionadas a críticas. Finalmente, as chamadas de métodos da *Thread 6* estavam relacionadas com a OCL e críticas e esta *thread* foi descartada pela mesma razão.

O plano de implementação do novo diagrama foi projetado por analisar chamadas de métodos contidas na *Thread 1* e *Thread 2*. O plano de implementação foi guiado por várias características que precisam ser adicionadas no *software*. Uma característica é um elemento observável do ponto de visão do usuário. As características projetadas são 1) item de menu para Criar Diagrama de Objetos, 2) botões para Adicionar Objetos em Diagramas e c) eventos para Adicionar uma Associação entre 2 Objetos. O plano de implementação consiste nos seguintes 3 passos descrevendo como cada característica deve ser implementada.

Passo 1: Criar Diagrama de Objetos. O primeiro ponto é encontrar onde os itens de menu são criados e então introduzir um novo item de menu para criar diagramas de objetos. A hipótese é que os segmentos alinhados na *Thread 1* mostre elementos comuns da inicialização do sistema. Portanto, algum destes elementos, por exemplo, o *menubar*, deveria ser adaptado para suportar o novo diagrama. A busca por chamadas de métodos relevantes iniciou com a expansão do primeiro nó da árvore, que tem dois filhos. A chamada de método `initializeSubsystems` da classe `Main` foi uma das selecionadas. Este método foi expandido - para o nível 2 - e o método `initializeGUI` da classe `Main` foi selecionada entre 11 nós porque o objetivo é compreender a inicialização da GUI. O processo de navegação foi o mesmo até o nível seis, onde foi encontrado a chamada de método `createApplicationMenuBar` da classe `MenuBarFactory` - o primeiro nó da Figura 3.4. Nesta chamada de método, o construtor da classe `MenuBar14` - nível 7 - foi selecionado após uma confirmação destas semânticas por inspeção no código fonte. As mudanças que deveriam ser feitas para inserir o menu “*New Object Diagram*” possivelmente seriam feitas no método `MenuBar14.initMenuCreate`, também mostrado na Figura 3.4. Este método foi inserido no plano para criar o item menu. Como mostra na árvore, existe uma instanciação da classe `ActionUseCaseDiagram.<init>` para o diagrama de casos de uso, então esta classe e método deve também ser adicionada ao plano porque assumimos que uma classe/método `ActionObjectDiagram.<init>` deveria também ser implementada.

Os métodos necessários para manusear o evento “Criar Diagrama de Objetos” foram encontrados no alinhamento entre o cenário A e B (*Thread 2*). A hipótese é que os seg-

mentos desalinhados durante o desenho de cada diagrama contém fragmentos específicos dos diagramas de classes e componentes e estes fragmentos deveriam ser também adicionados para o novo diagrama. A Figura 3.5 mostra um fragmento da árvore, distinguindo as chamadas específicas de cada cenário. Os métodos de cada cenário foram apresentados na árvore com cores diferentes, sendo mais fácil localizá-los. Por exemplo, a classe `ActionDeploymentDiagram` contém os métodos `actionPerformed`, `findNamespace`, `isValidNamespace`, `createDiagram` e a classe `ActionClassDiagram` contém os mesmos nomes de métodos. Então, estes métodos foram inseridos no plano para criar o diagrama de objetos. O método `create` apresentado no segmento da classe `ActionClassDiagram` foi inserido no plano também.

Passo 2: Adicionar Objetos em Diagramas. A seleção dos métodos que ajudariam a compreender como adicionar objetos em diagramas foi baseado na hipótese que segmentos desalinhados que contém chamadas de métodos relacionadas semanticamente com “3. Adicionar Classe C1” e “3. Adicionar Componente D1”, forneceriam informações relevantes para Adicionar Objeto. A busca por estes métodos foi feita da mesma maneira que o Passo 1. Os métodos que foram considerados semanticamente relacionados à adição do objeto em um diagrama foi `addNode`, `addElementListener`, `createDiagramElement`, `getFigNodeFor`, `canAddNode`, `createDiagram` e `doesAccept`.

Passo 3: Adicionar Associação entre 2 Objetos. A hipótese para este passo foi a mesma do passo anterior. Os segmentos desalinhados contendo chamadas de métodos relacionadas com “4. Inserir Associação entre C1 e C2” e “4. Inserir Dependência entre D1 e D2” foram analisadas porque o desalinhamento deveria prover informações relevantes para Adicionar Associação entre Objetos O1 e O2. Os métodos selecionados no plano foram `addEdge`, `canAddEdge`, `getFigEdgeFor`, `createAssociation`, `connect`, `createAssociationEnd`, `paint`, `createDependency`, `buildConnection`, `paintClarifiers`, `buildAssociation` e `buildDependency`.

ArgoUML - Análise de Resultados

Após o projeto do plano de implementação, um conjunto de 31 métodos foram listados por nossa abordagem como sendo importantes na condução da mudança perfectiva. O controle para avaliar a qualidade deste conjunto é o **Reais Positivos (RP)**, métodos que foram descobertos durante a implementação do novo diagrama. Um conjunto de 32 métodos foram realmente necessários para adicionar a característica. Os métodos incluídos neste conjunto são aqueles que foram necessários para compreender a mudança requerida e aqueles que foram efetivamente modificados. O diagrama de objetos foi implementado no ArgoUML com aproximadamente 606 linhas de código, incluindo inserções feitas pela IDE Eclipse tal como *imports*, construtores *stub*, métodos adicionados automaticamente após a inserção da cláusula *implements*, etc. A Tabela 3.4 mostra o número de elementos inseridos no código fonte.

A Tabela 3.5 mostra uma análise do resultado em termos de VP (Verdadeiros Positivos:

Tabela 3.4: Número de elementos inseridos no ArgoUML.

Tipos de Elementos	Quantidade
Pacotes	2
Classes	6
Métodos	13
Propriedades	3
Total de LOC inseridas	606

Tabela 3.5: Precisão e *Recall* no ArgoUML.

Sistema	VP	FP	FN	Precisão %	Recall%
ArgoUML	19	6	12	76	61,29

métodos apontados pela abordagem e que foram efetivamente usados), FP (Falsos Positivos: métodos apontados pela abordagem que não foram usados) e FN (Falsos Negativos: métodos que foram necessários mas não foram apontados pela abordagem).

3.3.2 Estudo de Caso 2 - Columba

Este estudo avalia a abordagem com um caso que requereu a abordagem ser aplicada duas vezes: para compreender onde a manutenção deveria ser feita e como reusar outras partes do sistema, ao invés de replicar a característica. As 3 fases da abordagem foram aplicadas como segue, com o objetivo de recuperar um conjunto de métodos para guiar a implementação da nova característica Enviar Mensagem Programada.

Definição e Execução dos Cenários

Dois cenários foram especificados para buscar por características existentes que eram similares à nova característica. A Tabela 3.6 apresenta uma correspondência próxima entre os passos dos cenários selecionados para atualizar o Columba.

Outro par de cenários foi selecionado para reusar algumas tarefas implementadas no sistema, mostrado na Tabela 3.7. O cenário “Novo Compromisso” foi selecionado para guiar em como reusar alguns métodos que inserem um novo compromisso no Calendário.

Tabela 3.6: Par de cenários para atualizar o Columba.

A. Enviar Mensagem Agora	B. Enviar Mensagem Depois
1. Inicializar o Sistema	1. Inicializar o Sistema
2. Escrever Nova Mensagem	2. Escrever Nova Mensagem
3. Enviar Agora	3. Enviar Depois

Tabela 3.7: Par de cenários para reusar código existente no Columba.

A. Enviar Mensagem Agora	C. Novo Compromisso
1. Inicializar o Sistema	1. Inicializar o Sistema
2. Escrever Nova Mensagem	2. Criar Calendário
3. Enviar Agora	3. Inserir Novo Compromisso

Sumarização dos Rastros de Execução

Durante a execução de cada cenário, dezessete arquivos de rastros foram capturados, um para cada *thread* disparada. O processo de compressão e sumarização foram o mesmo como no estudo de caso anterior. A Tabela 3.8 mostra o resultado de cada passo para reduzir os rastros dos três cenários selecionados.

Alinhamento dos Rastros Sumarizados

O método de alinhamento nas *threads* sumarizadas foi executado da mesma maneira que no estudo de caso anterior para produzir as árvores de alinhamento.

Definição do Plano de Implementação

Após a análise visual das árvores de alinhamento, métodos foram selecionados por análise cuidadosa dos seus nomes de pacotes, classes e métodos. *Threads* que continham chamadas de métodos relacionadas com “Enviar Mensagem Agora”, “Enviar Mensagem Depois” e “Inserir Novo Compromisso” foram identificados e os outros foram descartados. Vários segmentos desalinhados e *threads* puderam ser descartados porque eles não tinham nomes de métodos semanticamente relacionados a nenhuma das características estudadas. Por exemplo, os métodos `getBorderInsets` e `mouseEntered` não foram considerados semanticamente relacionados às características “Enviar Mensagem Agora” e “Enviar Mensagem

Tabela 3.8: Resultados da Compressão e Sumarização dos Rastros de Execução do Columba.

	Cenário A	Cenário B	Cenário C
Tarefas	3	3	3
Tamanho do Rastro Bruto	132.872	89.871	116.825
Tamanho do Rastro Comprimido	29.521	25.107	30.887
Taxa de Compressão	77,78%	72,06%	73,56%
Tempo de Execução(<i>ms</i>)	5376	3464	3831
Tam. do Rastro Comp-Sumarizado	7.025	6.012	6.785
Taxa de Compressão	94,71%	93,31%	94,19%
Tempo de Execução(<i>ms</i>)	904	778	834
Tam. do Rastro Comp-Sum-Comp	5.788	5.218	5.788
Taxa de Compressão	95,64%	94,19%	95,22%
Tempo de Execução(<i>ms</i>)	571	645	358

Depois”. Na verdade, subjetividade é um fato e esta decisão pode reduzir a métrica *recall*. Foi adotado uma abordagem conservativa, tal que, no caso de dúvida é preferível não descartar.

O plano de implementação da nova característica foi especificada analisando as chamadas de métodos nas *threads* 3, 4, 5, 8, 16 e 18. O plano de implementação é guiado pelas características requeridas para serem adicionadas e reusadas no *software*. O plano de implementação consiste dos 3 seguintes passos, descrevendo como cada característica tem que ser adicionada.

Passo 1: Adicionar Botão na Barra de Ferramentas. A seleção dos métodos que ajudariam a compreender como um botão está adicionado na barra de ferramentas foi baseada nos segmentos desalinhados da *Thread 3*. Esta *thread* continha as chamadas de métodos executadas para escrever uma nova mensagem, então ela foi uma candidata para informar onde o novo botão tinha que ser adicionado. Os botões e painéis no Columba são construídos usando arquivos XML. Então, alguns métodos específicos recuperados da árvore

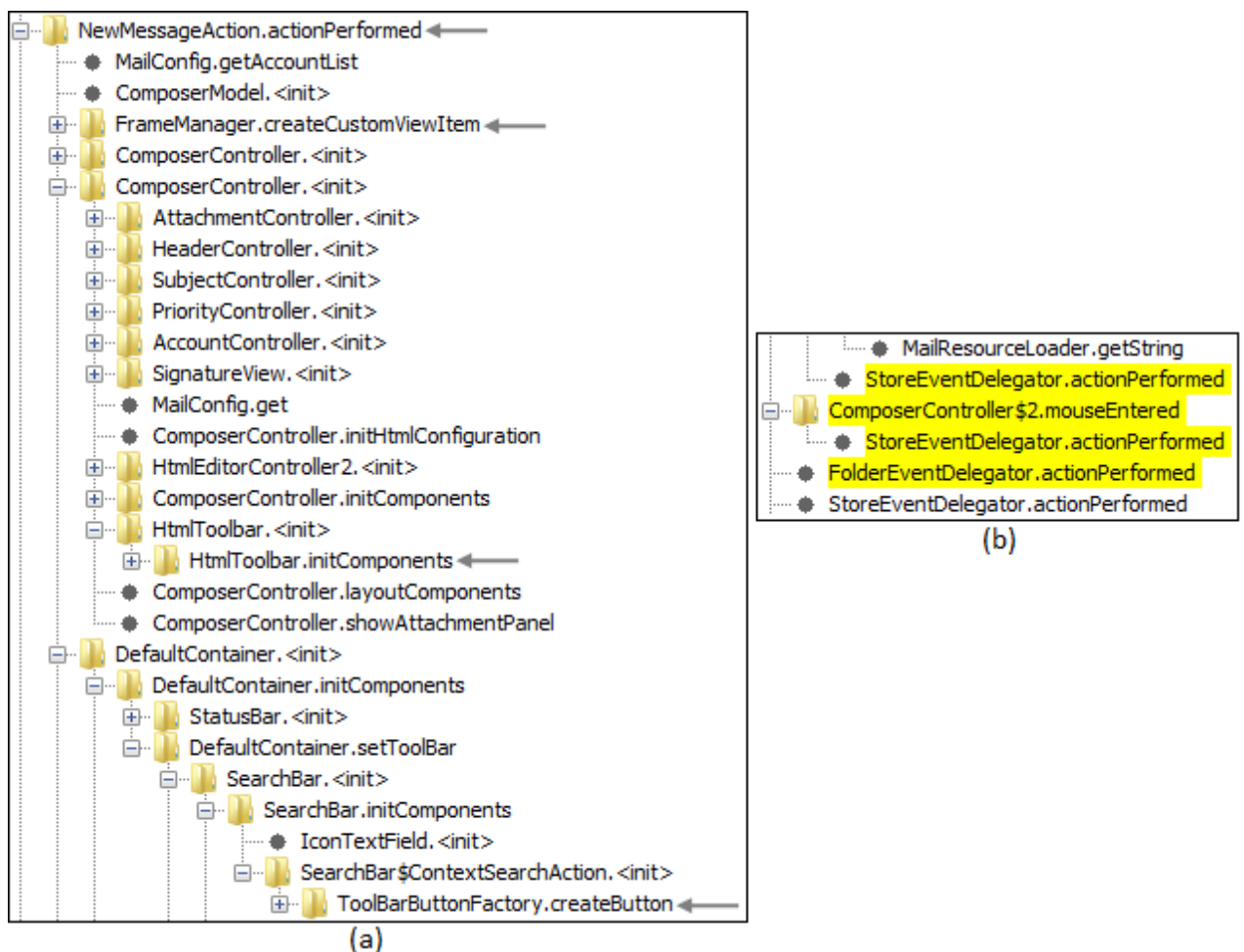


Figura 3.6: (a) Fragmento da árvore de alinhamento para detectar as chamadas de métodos para criar um botão na barra de ferramentas e (b) fragmento da árvore que contém desalinhamento.

de alinhamento puderam ser usados para marcar *breakpoints* em uma sessão de depuração para identificar qual arquivo era lido durante a criação da janela de nova mensagem. As chamadas de métodos consideradas semanticamente relacionadas com a criação do *frame* e a barra de ferramentas foram `FrameManager.createCustomViewItem`, `HtmlToolBar initComponents`, `ToolBarButtonFactory.createButton` e `NewMessageAction.actionPerformed`. Estes métodos foram inseridos no plano para adicionar o nome do novo botão no arquivo `XML`. As setas cinzas na Figura 3.6 mostram as chamadas de métodos selecionadas na árvore de alinhamento. Estas chamadas de métodos não foram desalinhadas porque o alinhamento da *Thread 3* da tarefa “2. Escrever Nova Mensagem” entre os dois cenários é o mesmo. Como as duas tarefas são as mesmas, o alinhamento das duas contribuiu na remoção de ruídos, como por exemplo, eventos de *mouse*. Para esta análise, os segmentos desalinhados tiveram outro significado. Eles apontaram ruídos no rastro, como mostra o fragmento da árvore na Figura 3.6.

Passo 2: Enviar Mensagem. A seleção dos métodos que ajudariam a compreender como uma mensagem é enviada foi baseada na hipótese que segmentos desalinhados, contendo chamadas de métodos semanticamente relacionadas a “3. Enviar Mensagem Agora” e “3. Enviar Mensagem Depois”, deveriam fornecer informação relevante. As chamadas de métodos que foram consideradas semanticamente relacionadas com envio de mensagem foram inseridas no plano de implementação como mostra a Tabela 3.9.

Passo 3: Novo Compromisso. A seleção dos métodos que ajudariam a compreender como um compromisso é criado foi baseado nos segmentos desalinhados que contém chamadas de métodos semanticamente relacionadas com “Inserir Novo Compromisso”. Os métodos selecionados, relacionados semanticamente com a característica, foram inseridas no plano de implementação. Estes métodos foram `createButtonPanel`, `createPanel` e `actionPerformed` da classe `NewAppointmentAction` e `CalendarPicker.init`.

Tabela 3.9: Conjunto de métodos do plano de implementação para a tarefa enviar mensagem no Columba.

Nome da Classe	Nome do Método
SendAction	actionPerformed
SendLaterAction	actionPerformed
SendMessageCommand	process
SendMessageCommand	execute
SaveMessageCommand	process
SaveMessageCommand	execute
CachedMboxFolder	addMessage

Tabela 3.10: Número de elementos inseridos no Columba.

Tipos de Elementos	Quantidade
Classes	2
Métodos	7
Inseção em propriedades e arquivos XML	3
Total de LOC inseridas	256

Columba - Análise de Resultados

Um conjunto de 14 métodos e 1 classe foram selecionados e inseridos no plano de implementação pela abordagem como sendo importantes para conduzir a mudança perfeita.

Os métodos **Reais Positivos** (*RP*) foram descobertos durante a implementação da nova característica no sistema. Um conjunto de 11 métodos foram realmente necessários durante o processo de adição da nova característica. A mensagem agendada foi implementada no Columba com aproximadamente 256 linhas de código. A Tabela 3.10 mostra o número de elementos inseridos no código fonte. A Tabela 3.11 mostra uma análise dos resultados em termos de VP, FP e FN.

Tabela 3.11: Precisão e *Recall* no Columba.

Sistema	VP	FP	FN	Precisão %	Recall%
Columba	11	3	0	78,6	100

3.3.3 Estudo de Caso 3 - Llama Chat

Este estudo avalia a abordagem com um caso onde a mudança perfeita também inclui o reuso de um sistema para herdar uma característica do outro. Neste caso, o sintetizador de fala FreeTTS⁴ foi reusado para inserir a síntese de fala no Llama Chat. A API fundamental do FreeTTS foi compreendida usando a documentação. As 3 seguintes fases foram executadas com o objetivo de recuperar um conjunto de métodos sugestivos que ajudariam a entender onde a característica do FreeTTS deveria ser introduzida no Llama Chat.

Definição e Execução dos Cenários

Este caso é diferente da abordagem do ArgoUML e do Columba porque não foi possível encontrar duas características semelhantes que fossem úteis para implementar uma terceira característica. Neste caso, os cenários puderam ser especificados com a busca de

⁴<http://freetts.sourceforge.net/docs/index.php>

Tabela 3.12: Par de cenários para o Llama Chat.

A. Enviar uma Mensagem Privada	B. Enviar Mensagem Depois
1. Inicializar o Sistema	1. Inicializar o Sistema
2. Mudar de Canal	2. Mudar de Canal
3. Enviar Mensagem para o Usuário B	3. Receber Mensagem do Usuário A

duas características, tal que um ou ambas estariam relacionadas aos métodos que potencialmente deveriam ser atualizados para suportar a nova característica. Neste estudo de caso, o objetivo era a composição de dois sistemas. No Llama Chat, um usuário A envia uma mensagem de texto privada ao usuário B e este recebe e ouve a mensagem de texto. Portanto, o ponto relevante de mudança é a parte do código que recebe uma mensagem, que deveria ser modificada para introduzir a síntese de fala. Os cenários selecionados, mostrados na Tabela 3.12, contém as características Receber Mensagem Privada e Enviar Mensagem Privada.

Sumarização dos Rastros de Execução

Durante a execução de cada cenário, quatro arquivos de rastros foram capturados, um para cada *thread* disparada. Em seguida, o processo de compressão e sumarização foi iniciado. Este processo foi o mesmo como os estudos de casos anteriores. A Tabela 3.13 mostra o resultado de cada passo para reduzir os rastros dos cenários selecionados.

Alinhamento dos Rastros Sumarizados

O método de alinhamento foi executado nos rastros sumarizados obtidos na Fase 2 e as árvores de alinhamento foram geradas da mesma maneira que os estudos de caso do ArgoUML e do Columba.

Tabela 3.13: Resultados da Compressão e Sumarização dos Rastros de Execução do Llama Chat.

	Cenário A	Cenário B
Tarefas	3	3
Tamanho do Rastro Bruto	262	116
Tamanho do Rastro Comprimido	145	96
Taxa de Compressão	30,29%	19,83%
Tempo de Execução(<i>ms</i>)	106	60
Tamanho do Rastro Comprimido-Sumarizado	27	22
Taxa de Compressão	89,69%	81,03%
Tempo de Execução(<i>ms</i>)	41	41
Tamanho do Rastro Comprimido-Sumarizado-Comprimido	20	21
Taxa de Compressão	92,37%	81,89%
Tempo de Execução(<i>ms</i>)	30	21

Definição do Plano de Implementação

Primeiro, uma análise visual das árvores de alinhamento ajudou a selecionar os métodos que seriam relevantes para a composição dos sistemas independentes. Os métodos foram selecionados por uma simples inspeção de nomes de pacotes, classes e métodos. O primeiro estudo das árvores de alinhamento mostrou que *threads* tinham chamadas de métodos relacionadas com envio de mensagem privada e recebimento de mensagem privada. As *Threads* 2 e 3 poderiam ser descartadas porque a *Thread* 2 continha somente chamadas de métodos para inicializar o sistema e a *Thread* 3, métodos para enfileirar as mensagens. O plano de implementação foi especificado com a análise das árvores de alinhamento da *Thread* 1 e *Thread* 4. O plano de implementação consistiu nos 2 seguintes passos:

Passo 1: Enviar Mensagem Privada. A seleção dos métodos que ajudariam a compreender como uma mensagem é enviada foi baseada no único segmento desalinhado da *Thread* 1. Os métodos selecionados, relacionados semanticamente a “3. Enviar Mensagem para o Usuário B”, foram `sendPrivate` e `newPrivate`. Eles foram inseridos no plano para adicionar a nova característica.

Passo 2: Receber Mensagem Privada. A hipótese para este passo foi o mesmo que o passo anterior. O alinhamento da *Thread* 4 continha três segmentos desalinhados. Os métodos selecionados, semanticamente relacionados a “3. Receber Mensagem do Usuário A”, foram `private_msg` e `recievePrivate`, mostrado na Figura 3.7. Eles foram inseridos no plano de implementação.

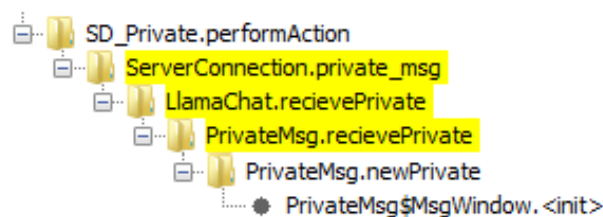


Figura 3.7: Segmento desalinhado referente ao recebimento de mensagem do Llama Chat.

Llama Chat - Análise de Resultados

O plano de implementação foi baseado em um conjunto de 4 chamadas de métodos. Os métodos **Reais Positivos** (*RP*) foram descobertos durante a implementação de comunicação dos dois sistemas. Um conjunto de 5 métodos foi realmente necessário para inserir a nova característica, incluindo todos os quatro métodos do plano de implementação. Os métodos incluídos são aqueles que foram necessários para compreender a mudança requerida. Somente um método foi efetivamente modificado. Doze linhas de código foram necessárias para incorporar a característica de reconhecimento de fala do FreeTTS dentro

Tabela 3.14: Precisão e *Recall* no Llama Chat.

Sistema	VP	FP	FN	Precisão %	Recall%
Llama Chat	4	0	1	100	80

do Llama Chat. A Tabela 3.14 mostra uma análise dos resultados em termos de VP, FP e FN.

3.4 Discussão

Um passo fundamental da abordagem de alinhamento de rastros sumarizados provou estar no processo de sumarização. O processo de sumarização é baseado na filtragem de métodos com baixa granularidade, por exemplo, métodos com operações primitivas ou métodos que chamam poucos métodos com baixa granularidade. Quanto maior a granularidade, maior o número de métodos podados do rastro. Poda excessiva é indesejável porque o alinhamento não forneceria informação suficiente para auxiliar na definição do plano de manutenção, então uma abordagem mais conservativa seria preferível ao preço de ter mais falsos positivos. Se o resultado do alinhamento provê informação excessiva, então o desenvolvedor pode usar valores de granularidades maiores para reduzir o tamanho dos rastros e consequentemente, o número de segmentos de alinhamento. Experimentos anteriores mostraram que a abordagem de alinhamento sem a sumarização dos rastros resulta em um número grande de segmentos desalinhados [Maia et al. 2008] e [Paixão 2009]. Esta situação introduziria um número grande de falsos positivos na abordagem, especialmente se fosse considerar uma abordagem conservativa para descartar nomes de métodos não relacionados.

Uma importante questão que poderia tentar ser respondida é “A abordagem pode ser considerada realmente efetiva em reduzir o esforço necessário para executar manutenção perfectiva?”. Supondo que um desenvolvedor precise inserir um diagrama no ArgoUML por depuração no código fonte. Quantas milhares de linhas deveriam ser depuradas? Não existe uma resposta definitiva para esta questão porque dependeria da habilidade do programador em não depurar dentro de métodos desinteressantes até ele encontrar métodos importantes que são necessários para compreender e implementar as modificações. Mas, as tabelas que descrevem o tamanho dos rastros mostram que mesmo para rastros sumarizados o número de chamadas de métodos é muito grande e poderia impôr um desafio significativo sobre a habilidade do programador em encontrar informação relevante. Por outro lado, se considerar os resultados da abordagem de sumarização com a técnica de alinhamento, altos níveis de precisão foram alcançados nos três estudos apresentados, 77%, 79% e 100%, respectivamente. Estes resultados significam que quase todos os métodos indicados como importantes na manutenção perfectiva foram de fato importantes. Além

disso, o *recall* dos estudos, 63%, 100% e 80%, respectivamente foram também significativos, porque mesmo se nem todas as informações são fornecidas, os métodos importantes que não são listados pela abordagem seriam fáceis de encontrar após o desenvolvedor ter tido uma compreensão melhor do sistema usando os métodos verdadeiros positivos sugeridos pela abordagem.

3.5 Considerações Finais

Neste capítulo foi proposta uma abordagem baseada na análise dos rastros de execução sumarizados com o objetivo de reduzir o esforço de tarefas de evolução de *software*. Este capítulo mostrou como recuperar informação importante do código fonte para ajudar a conduzir manutenção perfectiva.

Uma das hipóteses é que os segmentos desalinhados dos rastros de execução são pontos de implementação específica de uma característica em particular. Baseada nesta hipótese, a abordagem de sumarização com a técnica de alinhamento foi avaliada na evolução de três diferentes sistemas de *software* de código aberto. Primeiro, o diagrama de objetos foi implementado na ferramenta de modelagem UML ArgoUML. Segundo, um reconhecimento de fala no par de *chat* servidor/cliente para uso no Llama Chat. Terceiro, foi implementado a mensagem agendada no cliente de *email* Columba. Os resultados experimentais mostraram uma qualidade considerável da lista de métodos produzidas pelas duas técnicas como sendo importantes para conduzir mudanças perfectivas. Os resultados sugeriram que o cenário de exemplo descrito em que um desenvolvedor não familiarizado com um sistema alvo, deveria introduzir um novo tipo de diagrama na ferramenta de modelagem UML com um ou dois dias de trabalho é na verdade viável.

Capítulo 4

Extração de Visões de Alto Nível de Abstração

O objetivo deste capítulo é apresentar uma técnica de recuperação de informações importantes dos rastros de execução para auxiliar na redocumentação de sistemas existentes tanto com modelos estruturais quanto comportamentais. As informações são representadas em uma árvore simplificada, onde o analista pode inserir valores de filtros para simplificar a análise visual. A Figura 4.1 ilustra uma visão geral da abordagem proposta. O processo consiste de quatro fases:

- Definição dos Cenários de Execução
- Sumarização dos Rastros de Execução
- Análise dos Rastros Sumarizados
- Agrupamento da *Tree View*

4.1 Definição dos Cenários de Execução

No primeiro passo, o analista deve definir um cenário típico de execução, que consiste de algumas tarefas. A Tabela 4.1 mostra um exemplo de cenário de execução que utiliza a biblioteca de persistência de objetos *Prevayler*¹ para transferir fundos entre duas contas bancárias. Por exemplo, o interesse da execução deste cenário é obter informações específicas de como uma conta corrente é criada e como um depósito é efetuado. No próximo passo, para coletar os rastros de execução, o cenário definido deve ser executado no sistema instrumentado. Cada linha do arquivo de rastro corresponde a uma chamada de método com o nome do método qualificado, seu nível correspondente na pilha de execução (para construção da árvore de chamadas representada em uma *Tree View*) e o tempo da chamada. Após a coleta dos rastros de execução, os rastros são sumarizados e separados

¹<http://www.prevayler.org/wiki/>

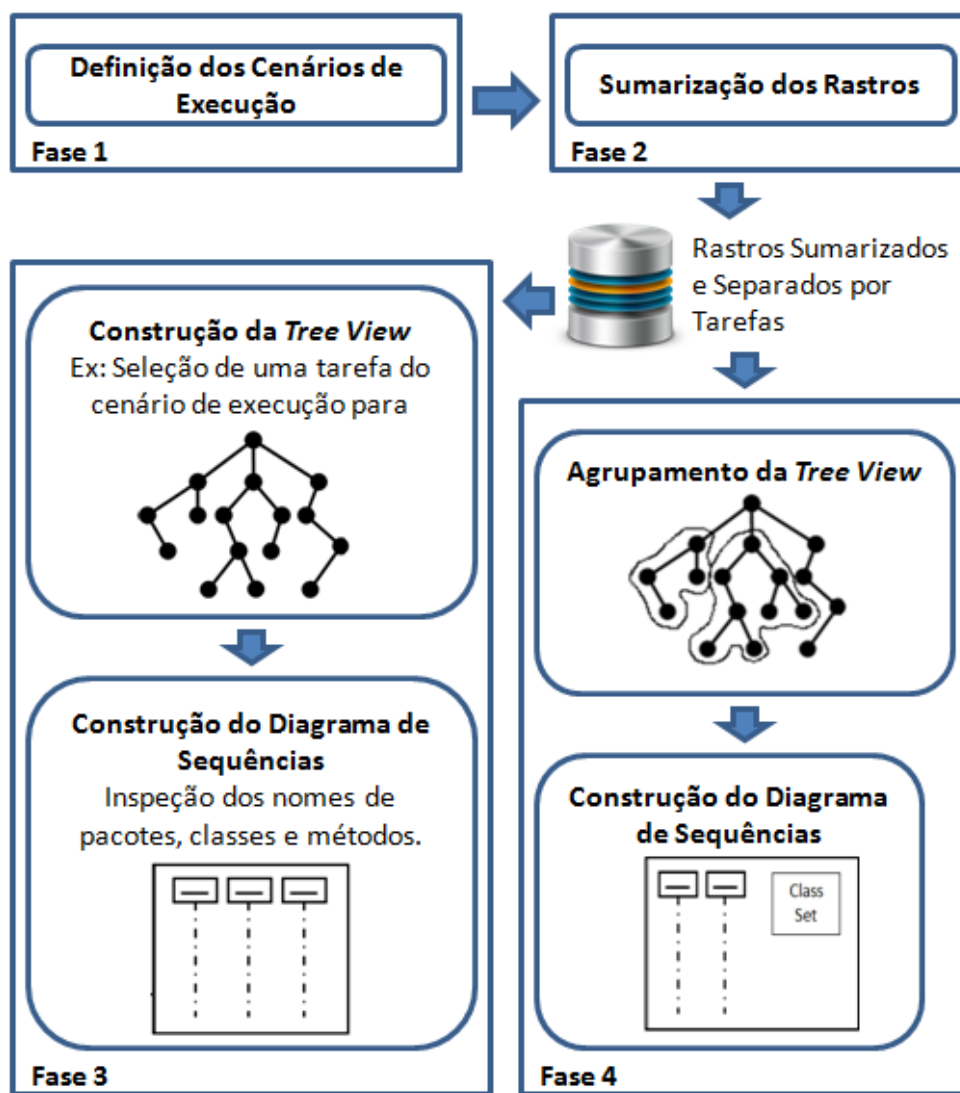


Figura 4.1: Descrição das 4 fases da técnica para recuperar visões de alto nível sobre rastros sumarizados.

Tabela 4.1: Cenário de Execução para o *Prevayler*.

Transferir Dinheiro
1. Inicializar o Sistema
2. Criar Conta Corrente C1
3. Depositar Dinheiro em C1
4. Criar Conta Corrente C2
5. Depositar Dinheiro em C2
6. Transferir dinheiro de C1 para C2

em arquivos de rastros, onde cada arquivo contém somente chamadas de métodos de uma tarefa do cenário.

4.2 Análise dos Rastros Sumarizados

Nesta fase, uma *Tree View* é gerada a partir do rastro de execução sumarizado para cada tarefa executada no cenário. A ferramenta de visualização dos rastros possui três filtros aplicados sequencialmente que afetam somente a *Tree View* ocultando os métodos da visualização. O sistema sugere valores *default* para os filtros, tal que não ocorra nenhuma filtragem adicional. Caso a visualização *default* não seja satisfatória, o analista pode alterar os valores dos filtros, ocultando chamadas com baixa granularidade. O primeiro parâmetro de filtragem é a profundidade máxima da árvore, que oculta os nós mais profundos. O valor inicial deste parâmetro é o valor mais profundo no rastro. Por exemplo, na Figura 4.2 a profundidade máxima desta árvore é 6. O segundo parâmetro de filtragem é o número máximo de nós por nível. Por exemplo, na Figura 4.2 o nó que possui

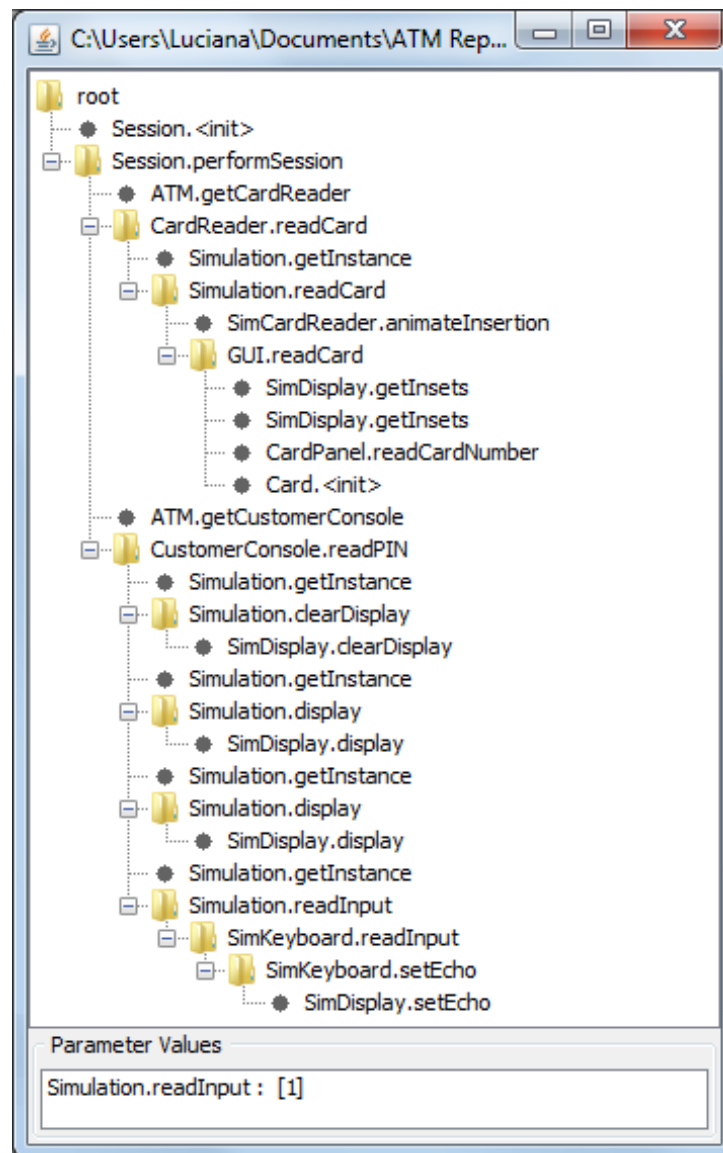
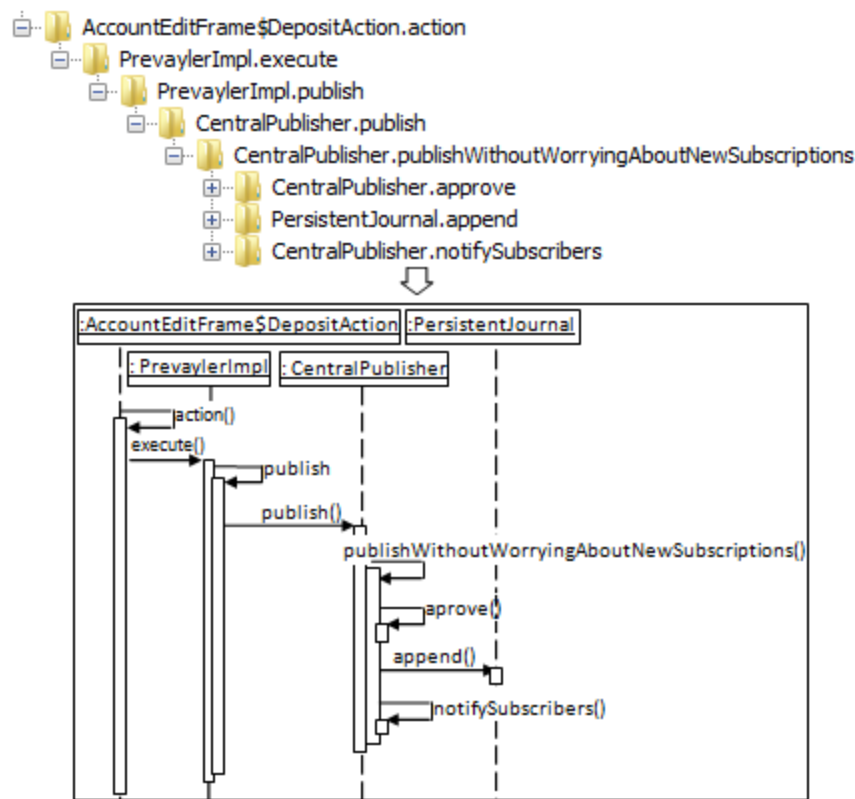


Figura 4.2: Árvore simplificada para extração de informação.

Figura 4.3: Transformação da *Tree View* para o diagrama de sequência.

o maior número de filhos é o `CustomerConsole.readPIN`. Este nó possui 8 filhos, então o valor *default* deste parâmetro é 8. O terceiro parâmetro de filtragem é a porcentagem mínima de granularidade de nós em um nível, considerando a árvore podada após a execução do terceiro filtro. O valor inicial deste parâmetro é zero e o usuário vai aumentando em uma unidade até que todas ou grande parte das chamadas de métodos com baixa granularidade sejam omitidas. Por exemplo, se em um nó arbitrário da árvore existem 5 nós filhos com granularidades cujos valores são 20, 20, 20, 21 e 19 e o parâmetro é 20, então o quinto nó é podado. Este último filtro geralmente é aplicável a árvores de rastros muito profundas, o que depende de cada sistema a ser avaliado. Nos estudos de casos apresentados neste artigo, as árvores tem vários filhos e não são muito profundas. Assim, este terceiro parâmetro não foi alterado, permanecendo o valor *default* da ferramenta. Na Figura 4.2 existe um campo abaixo da árvore que apresenta o conteúdo das mensagens de uma chamada de método selecionada, que na figura chamada selecionada é `Simulation.readInput`. A Figura 4.3 mostra a *Tree View* gerada pela ferramenta de visualização de rastros e o diagrama de sequências obtido a partir da *Tree View*. A característica selecionada é *Depositar Dinheiro em C1* do cenário do Prevayler apresentada na Tabela 4.1.

Por meio de análise da *Tree View* é possível recuperar diagramas de pacotes do sistema, porque cada nó, que representa uma chamada de método, contém as informações como nome do pacote, da classe e do método. Então, para obter uma visão geral da arquitetura do sistema, é possível agrupar vários diagramas de pacotes de várias execu-

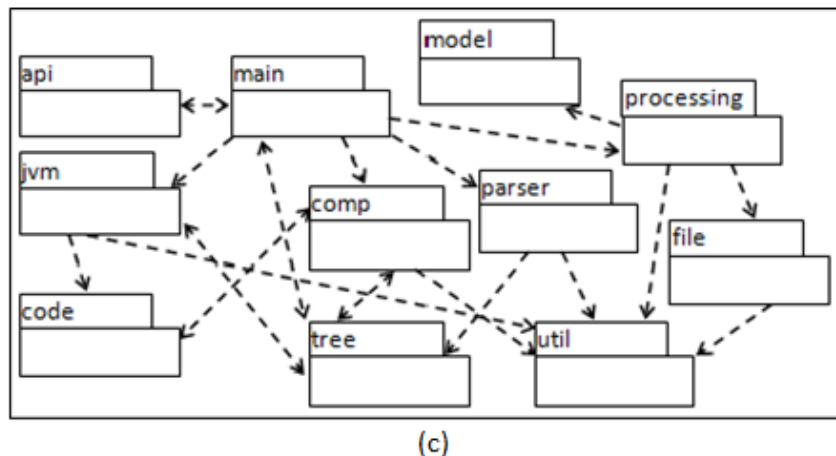
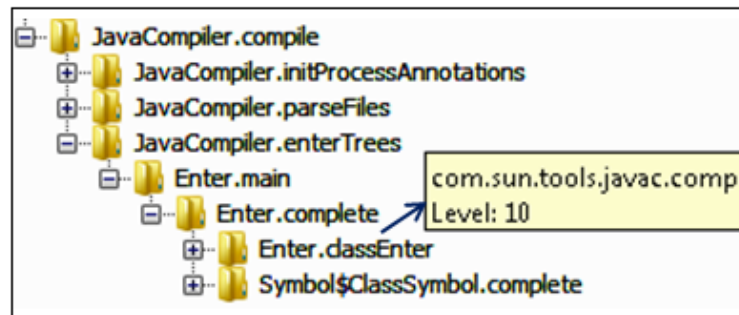
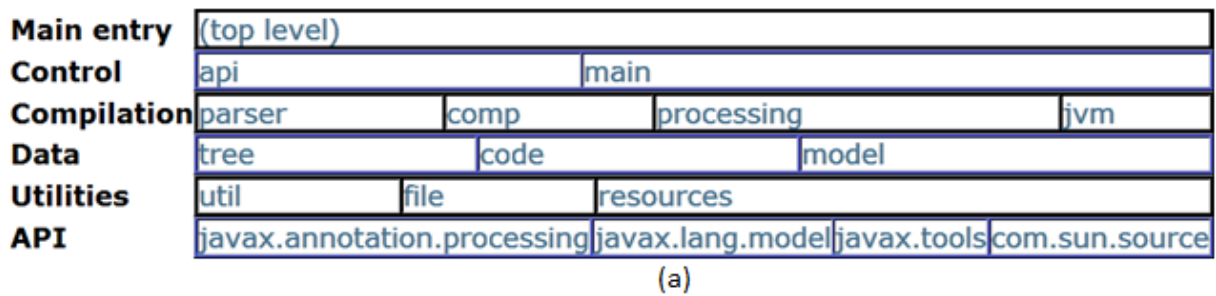


Figura 4.4: Transformação da *Tree View* para o diagrama de pacotes do Javac.

ções de diferentes cenários em um único diagrama. A Figura 4.4.c mostra o diagrama de pacotes recuperado dos rastros de execução do compilador Javac do OpenJDK² 6.0, uma implementação do compilador Java. A documentação disponibiliza os pacotes organizados em camadas, conforme mostra a Figura 4.4.a. A Figura 4.4.b mostra um fragmento da *Tree View* do Javac, permitindo extrair o nome dos pacotes e as dependências entre eles. Pode-se observar que na camada *Control*, na Figura 4.4.a, contém os pacotes *api* e *main* que possuem interface com a camada *Compilation*, que contém os pacotes *parser*, *comp*, *processing* e *jvm*. O diagrama de pacotes apresentado na Figura 4.4.c mostra estes pacotes e a dependência entre eles, inclusive entre as camadas. A partir de análises na *Tree View* foi possível detectar que o pacote *main*, da camada *Control*, faz acesso a classe(s) no pacote *tree* pertencente à camada *Data*. Esta informação mostra relação entre pacotes que estão

²<http://openjdk.java.net/>

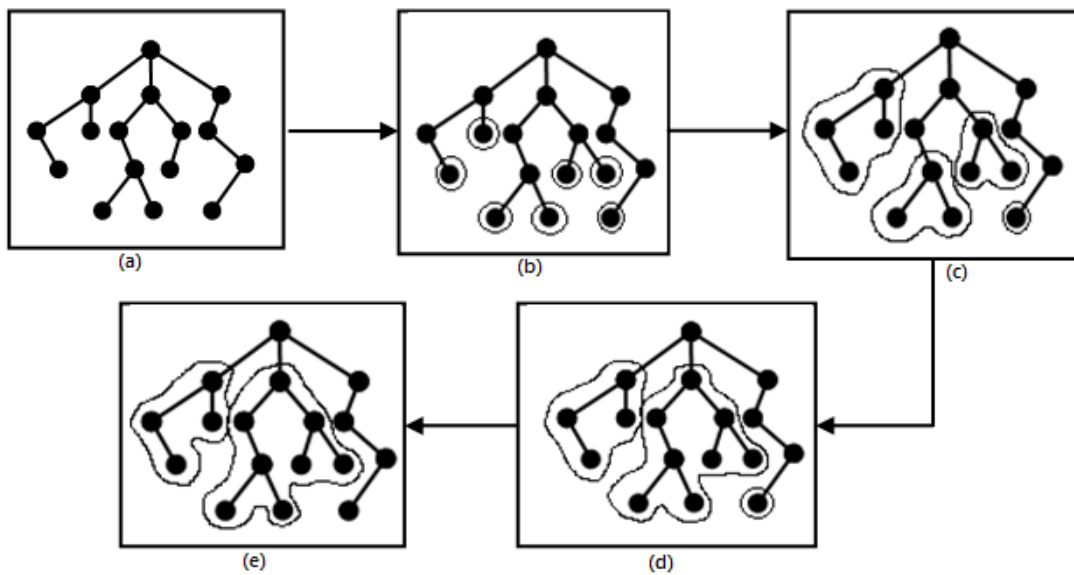


Figura 4.5: Fases de agrupamento do algoritmo.

em camadas que não são adjacentes uma à outra.

4.3 Agrupamento da *Tree View*

O diagrama de sequência pode ser construído com níveis de detalhes diferentes, de acordo com diferentes parâmetros selecionados na fase anterior. No entanto, em sistemas complexos o diagrama pode ser difícil de compreender porque mesmo após todo o processo de sumarização, o número de chamadas de métodos talvez ainda seja maior que o desejado. A razão é que o nível de abstração do código fonte talvez esteja muito detalhado para captar intenções de projeto. Por esta razão, o objetivo desta fase é agrupar sub-árvores para simplificar a visão do diagrama de sequências recuperado. Cada grupo de objetos das sub-árvores pode ser visto como um componente. A Figura 4.5 mostra o algoritmo para agrupar sub-árvores de maneira *bottom-up*. É fornecido como entrada o número M de componentes esperados. Seja T uma árvore de rastros. No primeiro passo, todos os nós folhas são selecionados e considerados como sub-árvores de componentes candidatos (Figura 4.5.b). Inicialmente, o número N de sub-árvores obtidas é igual ao número de folhas na árvore T . No próximo passo, cada nó folha n_f é agrupado com seu nó pai n_p se este possuir mais de 1 filho. Neste caso, o nó pai passa a ser a raiz da sub-árvore obtida. Neste caso, N necessariamente reduz (Figura 4.5.c). Nos próximos passos, n_p agrupa com seu nó pai se a mesma condição for satisfeita (Figura 4.5.d). O processo é executado até N não puder ser mais reduzido ou se N é igual ao M desejado. A Figura 4.5.e mostra as principais sub-árvores finais de componentes candidatos.

4.4 Estudos de Caso e Avaliação dos Resultados

O objetivo desta seção é avaliar a qualidade de diagramas de sequências recuperados pela abordagem proposta em dois sistemas de código aberto. Estes sistemas foram selecionados com o critério de possuir diagramas de sequências acessíveis, para que a qualidade dos resultados obtidos pudessem ser avaliados. O primeiro *software* a ser avaliado é o sistema de simulação ATM³ que possui 24 classes. Uma implementação similar em C++ do mesmo sistema foi usado para validar a abordagem proposta em [Briand et al. 2003]. O sistema possui documentos que inclui casos de uso que são necessários para definir os cenários. Possui também os diagramas de sequência correspondentes aos diferentes cenários. Com estas informações é possível comparar os resultados obtidos pela abordagem proposta. O segundo sistema selecionado é o Tomcat⁴ 6.0, uma implementação das tecnologias JavaServlet e JavaServer Pages com aproximadamente 163 KLOC. Na sua documentação possui uma visão geral da arquitetura do Tomcat, descrição detalhada de diagramas de sequência de como o Tomcat é inicializado e como ele lida com uma solicitação. Os passos definidos da Seção 2 foram seguidos para reduzir a complexidade dos rastros e extrair informação suficiente para gerar diagramas de sequências de alto nível, para cada tarefa em um cenário de execução. Após a construção dos diagramas de sequência é avaliada a relevância das chamadas selecionadas durante a fase de análise dos rastros.

4.4.1 Avaliação dos Resultados

As chamadas apontadas como relevantes pela nossa abordagem foram baseadas em suas granularidades. Os nós de chamadas “Reais Positivos”(RP) usados como controle são aqueles contidos na documentação e que estão também contidos no rastro de execução de um determinado cenário. Para validar os reais positivos foi feito um estudo nos documentos, juntamente com rastros de execução e o código fonte para compreender se os métodos que estão na documentação ainda existem no código e se existem, compreender o motivo da ausência no rastro, possivelmente por desuso. Com a sequência RP definida é possível calcular os seguintes valores que são usados para avaliar a abordagem em termos de precisão⁵ e *recall*⁶.

VP: Verdadeiros Positivos. Nós de chamadas que foram apontadas por nossa abordagem e estes estão contidos no conjunto RP.

FP: Falsos Positivos. Nós de chamadas apontadas por nossa abordagem, mas não necessárias, pois não estão em RP.

³www.math-cs.gordon.edu/local/courses/cs211/ATMExample/

⁴<http://tomcat.apache.org/tomcat-6.0-doc/architecture/index>

⁵É a proporção dos métodos relevantes que o sistema selecionou, ex.: $VP/(VP + FP)$

⁶É a proporção dos métodos selecionados que o sistema selecionou corretamente, ex.: $VP/(VP + FN)$

Tabela 4.2: Cenário de Execução para o Simulador ATM

A. Transferir Dinheiro	B. Consultar Saldo
1. Inicializar o Sistema	1. Inicializar o Sistema
2. Acionar ATM	2. Acionar ATM
3. Inserir Cartão	3. Inserir Cartão
4. Inserir PIN	4. Inserir PIN
5. Escolher Tipo de Transação	5. Escolher Tipo de Transação
6. Conta para Transferir de	6. Escolher Consulta do tipo de conta
7. Conta para Transferir para	
8. Valor da Transferência	

FN: Falsos Negativos. Nós de chamadas que estavam contidos no conjunto RP mas não foram apontados por nossa abordagem.

4.4.2 Análise do Sistema de Simulação ATM

A seguir são descritas as 4 fases da abordagem para recuperar os diagramas de sequências e compará-los com os diagramas existentes na documentação.

Definição e Execução dos Cenários

O simulador ATM possui as seguintes características: *Depositar Dinheiro*, *Transferir Dinheiro*, *Sacar Dinheiro* e *Consultar Saldo*. As 4 características do sistema possuem diagramas de sequência e de comunicação que servem de base para analisar a qualidade dos resultados. A Tabela 4.2 mostra dois cenários selecionados para avaliar a qualidade dos diagramas recuperados.

Tabela 4.3: Resultados da Compressão e Sumarização dos Rastros de Execução do Simulador ATM.

	Cenário A	Cenário B
Tarefas	8	6
Tamanho do Rastro Bruto	449	373
Tamanho do Rastro Comprimido	306	271
Taxa de Compressão	31,85%	27,34%
Tempo de Execução(ms)	531	203
Tamanho do Rastro Comprimido-Sumarizado	123	109
Taxa de Compressão	72,61%	70,78%
Tempo de Execução(ms)	78	31
Tamanho do Rastro Comprimido-Sumarizado-Comprimido	121	107
Taxa de Compressão	73,05%	71,31%
Tempo de Execução(ms)	32	31

Sumarização dos Rastros de Execução

Para cada cenário executado, quatro arquivos de rastros foram coletados, um para cada *thread* disparada. O compressor foi executado sobre os rastros para remover os laços e as chamadas recursivas. A sumarização foi feita manualmente, mantendo as chamadas de métodos com granularidade maior que 1, ou seja, somente foram removidas as folhas das árvores. Na tentativa de remover chamadas de métodos com granularidades maiores ou iguais a 2, métodos específicos com as características foram removidas, e por isto usou-se a sumarização anterior. Finalmente, o compressor foi executado sobre os rastros sumarizados para remover repetições que não foram detectadas na primeira compressão. A Tabela 4.3 mostra o resultado de cada passo para reduzir os rastros de cada cenário. A taxa de redução não ultrapassou 90% porque o sistema é razoavelmente pequeno.

Análise dos Rastros Sumarizados

Para as duas características, as *threads* usadas para gerar os diagramas de sequência foram as *Threads* 3 e 4. Esta seleção foi feita com base na separação das tarefas, apresentada no Capítulo 2, cuja informação foi obtida durante a execução dos cenários que permite identificar quais *threads* e chamadas de métodos foram usadas para executar uma determinada tarefa. Os métodos para criar os diagramas de sequência foram selecionados por uma simples verificação de nomes dos métodos, classes e pacotes e a granularidade do método analisado. Nos rastros do ATM, *Session*, *Transaction* são classes potencialmente importantes no sistema porque seus métodos aparecem em um nível mais alto da *Tree View* com granularidade alta. Essa afirmação é confirmada com análise na documentação. Antes de executar todas as tarefas contidas nas características do sistema, primeiro é criada uma sessão, seguida da criação de uma transação.

Transferir Dinheiro. Durante a análise da *Tree View*, 17 chamadas de métodos foram selecionadas para a criação do diagrama de sequência. No diagrama do simulador ATM contém uma sequência de 22 métodos referentes a este cenário. Portanto, o $RP = 22$ para esta característica.

Consultar Saldo. Através da *Tree View*, 16 chamadas de métodos foram selecionadas para a criação do diagrama de sequência. No diagrama do simulador ATM contém uma sequência de 21 métodos referentes a este cenário. Assim, o valor de $RP = 21$. A Figura 4.6 mostra o diagrama de sequência recuperado a partir dos rastros descrevendo como é feito a criação de uma sessão no sistema relacionados à consulta de saldo na conta.

Agrupamento da Tree View

As sub-árvores na *Tree View* foram detectadas através da execução do algoritmo *GroupTreeView* apresentado na Seção 2. Como os diagramas recuperados são pequenos, não há necessidade de simplificá-los. Os componentes candidatos recuperados são *Transfer*,

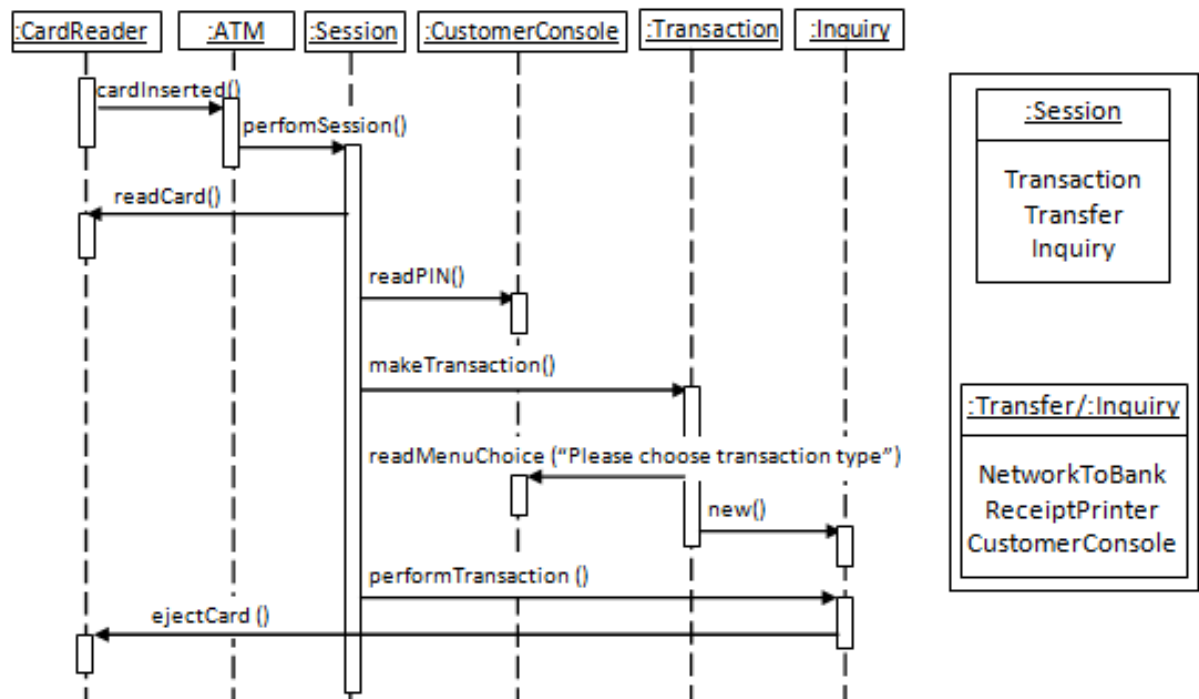


Figura 4.6: Diagrama de sequência de Consulta de Saldo do Simulador ATM e seus componentes encontrados.

Transaction, Inquiry, NetworkToBank, ReceiptPrinter e CustomerConsole. O componente principal é a classe Session, como mostra a Figura 4.6.

ATM - Análise dos Resultados

Após a seleção dos métodos para constituírem os novos diagramas foi feita uma avaliação de qualidade destes métodos com os métodos contidos na sequência dos Reais Positivos - RP. A sequência de métodos selecionada pela abordagem dos rastros referentes à característica *Transferir Dinheiro* contém 17 métodos e a sequência RP contém 22. A característica *Consultar Saldo* contém 16 métodos e a sequência RP contém 21. A Tabela 4.4 mostra uma análise dos resultados em termos de VP, FP e FN. Pode-se observar que a precisão para ambas características foi de 100% e que o *recall* foi de aproximadamente 76%. $FN = 5$ para os dois cenários e os métodos são os mesmos. Um estudo no código fonte destes 5 métodos foi feito para identificar a importância relativa deles na compreensão de cada característica. Três são somente construtores das classes Message, Receipt e Session, não tendo impacto na tarefa de compreensão. Os métodos RP que fazem parte da inicialização do sistema e que não foram apontados pela abordagem são `CashDispenser.setInitialCash` e `NetworkToBank.openConnection`. O primeiro método faz uma operação de atribuição do conteúdo recuperado pelo método `OperatorPanel.getInitialCash` que verifica a quantidade de dinheiro na máquina. O segundo método é um *stub*, não possui nenhuma linha de código dentro dele. Logo, sua granularidade é zero por este motivo

Tabela 4.4: Precisão e *Recall* para as duas características do simulador ATM.

Característica	VP	FP	FN	Precisão	<i>Recall</i>
Transferir Dinheiro	17	0	5	100%	77,3%
Consultar Saldo	16	0	5	100%	76,2%

foi removido pelo algoritmo de sumarização. Se considerarmos um espaço de busca de 373 chamadas originalmente, a precisão obtida com os VP pode ser considerada significativa.

4.4.3 Análise do Tomcat

As 4 fases a seguir foram executadas com o objetivo de recuperar métodos importantes para a construção dos diagramas de sequências.

Definição e Execução dos Cenários

Para avaliar os diagramas recuperados do Tomcat, o cenário selecionado consistiu de duas tarefas existentes: *inicialização do servidor* e o *acesso de dois usuários a uma única página JSP de exemplo*, porque existem diagramas de sequência destas tarefas na documentação do sistema para permitir a avaliação.

Sumarização dos Rastros de Execução

Durante a execução do cenário, dez arquivos de rastros foram coletados, um para cada *thread* disparada. O processo de sumarização é similar ao do estudo de caso anterior. Chamadas de métodos com granularidades menores ou iguais a 2 foram removidas, obtendo uma redução no volume dos rastros em mais de 90%. A Tabela 4.5 mostra o resultado de cada passo para reduzir os rastros do cenário selecionado.

Tabela 4.5: Resultados da Compressão e Sumarização dos Rastros do Tomcat

	Bruto	Comprimido	Comp-Sumarizado	Comp-Sum-Comp
Tamanho do Rastro	854.255	267.931	32.961	21.651
Taxa de Compressão	-	68,63%	96,14%	97,46%
Tempo de Execução(ms)	-	98.344	3.625	1.068

Análise dos Rastros Sumarizados

Inicialização do Servidor. Para criar o diagrama de sequência foram selecionadas as *threads* 1, 2, 3 e 6. Esta informação foi obtida da separação de tarefas na fase anterior. Os métodos foram selecionados por uma simples inspeção de nomes dos métodos, classes e pacotes e considerando também a granularidade do método que está sendo analisado.

Nos rastros do Tomcat, Catalina, Bootstrap e StandardServer são classes potencialmente importantes no sistema porque seus métodos aparecem no nível mais alto da *Tree View* com granularidade alta. Em adição, os nomes das classes e métodos geralmente tem um nome relacionado com a tarefa analisada. Por exemplo, a classe Catalina contém os métodos: `main`, `process`, `load` e `start`. Estes nomes de métodos estão ligados com a inicialização do servidor. A *Thread 1* foi selecionada para construir o diagrama de sequência porque contém o `Catalina.start` e a documentação do Tomcat disponibiliza um diagrama similar, tornando a comparação possível. O diagrama de sequência recuperado pela abordagem não está apresentado neste artigo por falta de espaço devido ao seu tamanho. Durante a análise da *Tree View*, 42 métodos foram selecionados para a criação do diagrama de sequência de inicialização do sistema. A documentação do Tomcat contém o diagrama de sequência referente à inicialização do servidor com uma sequência de 90 métodos. Estes métodos foram verificados juntamente com o rastro de execução e o código fonte. Dos 90 métodos na documentação, somente 51 métodos estão de fato presentes no rastro sequencialmente. Portanto, consideramos o RP de inicialização igual 51. Tanto a sequência de métodos selecionados para a criação do novo diagrama quanto a sequência de métodos presente na documentação contém métodos repetidos, porque para executar a inicialização do sistema alguns métodos são executados mais de uma vez. Três (3) classes e quatorze (14) métodos deixaram de existir na versão 6.0 do Tomcat.

Acesso de 2 Usuários a uma Página JSP. Para o diagrama de sequência foi selecionado a *Thread 9*. Com a separação de tarefas na fase anterior foi detectado que somente esta *thread* foi executada no momento em que um usuário acessa uma página JSP. Os métodos foram selecionados da mesma maneira que para a tarefa anterior. Nos rastros do Tomcat, `Http11Processor`, `CoyoteAdapter`, `StandardEngineValve`, `StandardHostValve`, `StandardContextValve`, `StandardWrapperValve`, `ApplicationFilterFactory` e `ApplicationFilterChain` são classes potencialmente importantes para esta tarefa porque aparecem em um nível alto da *Tree View* com granularidade alta. Os nomes dos métodos também estão relacionados com a tarefa analisada. Por exemplo, `InternalInputBuffer.parseHeaders`, `Http11Processor.prepareRequest`, `CoyoteAdapter.service`, `CoyoteAdapter.postParseRequest`. Um diagrama de sequência que descreve o processo de solicitação está disponível na documentação do Tomcat, então é possível avaliar os resultados obtidos na aplicação da abordagem. Durante a análise da *Tree View*, 15 métodos foram selecionados para a criação do diagrama de sequências mostrando como é feito a solicitação de uma página JSP. A documentação do Tomcat contém uma sequência de 35 chamadas de métodos para esta tarefa. Estes métodos foram analisados juntamente com o rastro de execução e o código fonte. Dos 35 métodos, somente 14 estão de fato presentes no rastro. Portanto, consideramos $RP = 14$ para esta tarefa. Três (3) classes e dez (10) métodos deixaram de existir na versão 6.0 do Tomcat.

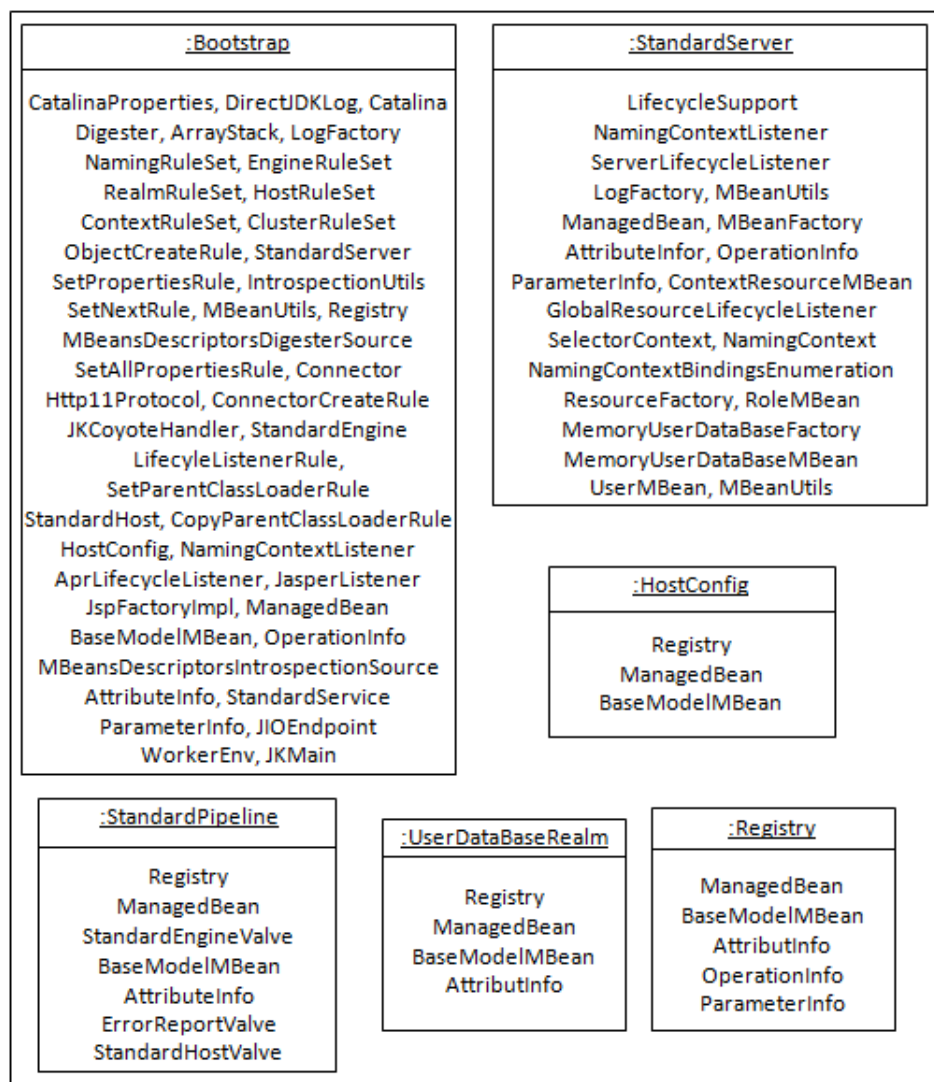


Figura 4.7: Componentes detectados pelo algoritmo de agrupamento da *Tree View* do Tomcat.

Agrupamento da *Tree View*

As sub-árvores na *Tree View* foram identificadas através da execução do algoritmo *GroupTreeView*. A lista de objetos do diagrama obtido na fase anterior foi relativamente grande e suas respectivas classes foram agrupadas dentro de componentes candidatos: descritos na Figura 4.7: *Bootstrap*, *StandardServer*, *HostConfig*, *StandardPipeline*, *Registry* e *UserDataBaseRealm*. Ao invés de descrever todas as classes no mesmo nível do diagrama, tornando o diagrama de sequência com excesso de informação, o diagrama foi simplificado agrupando várias classes dentro de componentes. Cada componente contém classes, cujos métodos de objetos invocados para uma característica específica foi encontrada pelas sub-árvores. Por exemplo, a classe *StandardServer* presente no diagrama principal contém 21 classes executadas a partir desta classe como mostra a Figura 4.7 e o componente *Bootstrap* contém 45 classes. Os componentes forneceram uma simplificação do diagrama

de sequência original, reduzindo o esforço para compreender a inicialização do Tomcat porque mostra uma visão de alto nível e ainda possibilita o acesso a chamadas de métodos intermediários.

Tomcat - Análise dos Resultados

Após selecionar os métodos que constituíram o novo diagrama é feito uma avaliação de qualidade destes métodos de acordo com os métodos contidos na sequência dos Reais Positivos - RPs.

Inicialização do Servidor. A sequência de métodos selecionada pela abordagem contém 42 métodos e a sequência RP contém 51. A Tabela 4.6 mostra uma análise dos resultados em termos de VP, FP e FN. A precisão reflete diretamente no nível de detalhamento do diagrama de sequência obtido pela abordagem em relação ao da documentação, ou seja, quanto mais baixa a precisão, mais detalhado estará o diagrama. Se diminuir a granularidade na fase de sumarização, os FP aumentam, diminuindo a precisão mas o *recall* aumenta porque os FN diminuem. Como ocorreram alguns FN, são apresentadas somente chamadas presentes na execução da tarefa na documentação e no rastro bruto, mas não presentes no rastro sumarizado. Estes métodos removidos pelo *Summarizer* possuem granularidade zero, por esta razão foram removidos por serem nós folhas na árvore. Um estudo foi feito destes métodos juntamente com o código fonte. O primeiro método que não estava presente na *Tree View* é o `Catalina.initDirs`. Este método é responsável por ajustar propriedades como `catalina.home` e `catalina.base`. Os seguintes métodos descritos executam atribuição de valores booleanos. O método `HostConfig.setDeployXml` ajusta a implantação do componente no arquivo de configuração XML. O método `HostConfig.setUnpackWars` ajusta o *flag* de WARs descompactados e `HostConfig.setXmlValidation` ajusta o *flag* da característica de validação do analisador XML usado quando analisar instâncias XML. Estes métodos descrevem em detalhes a execução de uma tarefa durante a inicialização do servidor. A nossa abordagem ajuda a compreender uma característica específica em um alto nível de abstração. Para estes métodos estarem presentes na *Tree View* seria necessário manter os métodos com granularidade zero. O problema é que os rastros seriam volumosos, dificultando a extração dos métodos VP. Consequentemente, a precisão seria menor que a apresentada na Tabela 4.6.

Acesso de 2 Usuários a uma Página JSP. A sequência de métodos selecionada

Tabela 4.6: Precisão e *Recall* para as duas tarefas do Tomcat.

Tarefas	VP	FP	FN	Precisão	<i>Recall</i>
Inicialização do Servidor	29	13	22	69,05%	56,86%
Acessar Página JSP	12	3	1	80,0%	92,31%

pela abordagem contém 15 métodos e a sequência RP contém 13. A Tabela 4.6 apresenta uma análise dos resultados em termos de VP, FP e FN. Um estudo dos métodos FNs foi feito no código fonte para verificar a importância relativa desses métodos. O primeiro `StandardContextValve.invoke` está presente no rastro sumarizado, foi visualizado na *Tree View*, foi selecionado para criar diagrama, mas não foi considerado VP por não estar na mesma posição da sequência de chamadas de métodos contidos na documentação. O segundo método `StandardWrapper.invoke` não está presente no rastro de execução bruto, por alguma razão na execução do cenário este método não foi executado.

4.5 Considerações Finais

Neste capítulo foi apresentada uma abordagem semi-automatizada baseada em análise dinâmica com uma visualização gráfica para facilitar a recuperação de informação com alto nível de abstração. A técnica apresentada foi baseada na relevância das chamadas baseada em um conceito de granularidade e no nível das mesmas na árvore visual gerada a partir dos rastros sumarizados. A técnica foi avaliada em dois sistemas de código aberto diferentes. O processo para recuperar informação através dos rastros foi dividida em quatro fases. Primeiro, os cenários são definidos e executados. Segundo, os rastros são sumarizados retirando chamadas de métodos com granularidade baixa. Terceiro, uma análise sobre os rastros sumarizados foi feita para selecionar os métodos considerados importantes para a compreensão de uma tarefa. E finalmente, um agrupamento é feito em modo *bottom-up* na árvore que sugere possíveis componentes. Estes componentes candidatos são uma maneira de simplificar o diagrama de sequência que está sendo construído. O processo é simples por não requerer configuração complexa e também é eficiente devido a execução que pode ser razoavelmente rápida em qualquer computador de *prateleira*. Os resultados de precisão e *recall* são expressivos considerando o vasto espaço de busca, além do que o alvo em termos de correção não necessariamente precisa ser o que consta literalmente na documentação utilizada como referência.

Embora os resultados sejam bastante promissores, existem melhorias a serem feitas. Por exemplo, chamadas de métodos do Tomcat que eram importantes para a compreensão foram perdidos pelo *Summarizer*.

Capítulo 5

Trabalhos Correlatos

Este trabalho tem como objetivo ajudar analistas e desenvolvedores de sistemas de *software* a compreenderem o conteúdo de rastros de execução extremamente volumosos. Neste capítulo serão apresentados os trabalhos relacionados com os seguintes tópicos: Sumarização de Rastros de Execução e Compreensão de Sistemas de *Software*.

5.1 Sumarização de Rastros de Execução

A sumarização de rastros reduz a complexidade dos mesmos, facilitando a extração de visões de alto nível para a compreensão das tarefas de um cenário de execução. Várias técnicas e algoritmos de compressão de rastros têm sido propostos. Todos são baseados no mesmo princípio: remoção de redundância. [Hamou-lhadj e Lethbridge 2002] propuseram um algoritmo de compressão de rastros que consiste na remoção de laços e chamadas recursivas. O número de ocorrências de cada redundância é armazenado para que seja possível reconstruir o rastro original. Este algoritmo de compressão é usado na abordagem proposta deste trabalho como um pré-processamento dos rastros para em seguida aplicar a nova abordagem de sumarização de rastros.

A técnica de sumarização de rastros proposta em [Hamou-Lhadj et al. 2005], [Hamou-Lhadj e Lethbridge 2006] descreve um algoritmo baseado na ideia de sumarização de textos e propõe que a sumarização de rastros recebe um rastro como entrada e retorna como saída um resumo dos principais eventos executados. O algoritmo classifica o componente, decidindo se é um componente chave ou um componente de utilidade, baseado em uma métrica de *fan-in/fan-out*. O objetivo é o mesmo da abordagem proposta neste trabalho que busca remover chamadas de métodos de utilidade. No entanto, neste trabalho, a definição do que é um componente relevante se baseia na noção de granularidade das chamadas de métodos.

[Myers et al. 2010] propuseram um algoritmo que compacta laços presentes em rastros de execução usando dados de múltiplas fontes: rastros, código fonte e informação de depuração. O objetivo é reduzir laços contidos em rastros de programas e apresentar

estes laços em diagramas de sequência. Os diagramas recuperados são muito locais e não mostram a relação de muitas classes relacionadas como a deste trabalho. Mas, este trabalho não fornece informação para produzir laços nos diagramas de sequências gerados.

A ferramenta de visualização *OutlineView* proposta por [Prada-Rojas et al. 2010] tem como objetivo sumarizar rastros de execução de sistemas de *software* embutidos através de uma visão compacta. A ferramenta simplifica a navegação em rastros volumosos por sumarizar características chaves. Uma visão compacta fornece um resumo dos rastros de execução a partir de uma dada perspectiva, escolhida pelo usuário final. *OutlineView* ajuda o usuário a encontrar mais facilmente defeitos ou melhorar a utilização de recursos em uma plataforma embutida. A abordagem de sumarização apresentada neste trabalho busca remover possíveis chamadas de métodos irrelevantes para a compreensão de sistemas orientados a objetos de um determinado cenário com várias tarefas.

[Kuhn e Greevy 2006] introduziram uma técnica de sumarização de rastros baseada na visualização de rastros completos em uma única janela. Eles mostram que em uma janela foi possível visualizar dezoito rastros sumarizados. Através da visualização foi possível observar quais características exibem padrões comuns de comportamento. A sumarização é baseada no nível de chamadas aninhadas em subsequências de chamadas de métodos, as informações como nome dos métodos, classes e pacotes são descartados durante o processo de sumarização. Este trabalho mostra que além de ser possível observar quais características exibem padrões comuns de comportamento, também é possível visualizar quais são os métodos e classes específicas e comuns entre essas características. Mas, a abordagem não permite comparar vários rastros de execução simultaneamente, somente em pares.

A abordagem proposta por [Taniguchi et al. 2005] tem como objetivo extrair diagramas de sequência compactos dos rastros de execução de programas orientados a objetos. O método propõe quatro regras de compactação para reduzir o volume de informação dos rastros. As regras compactam o rastro de execução por abstrair alguns padrões de repetições e chamadas recursivas incluídas no rastro. Após o rastro ser compactado é então traduzido para um diagrama de sequência compactado. As regras detectam uma repetição de subárvores similares em uma *call tree* e as substituem com uma representante mostrando toda a estrutura e o número de repetições. No rastro compactado, todas as chamadas presentes no rastro permanecem, descrevendo detalhadamente a execução de uma característica. O trabalho proposto por [Delamare et al. 2006] é semelhante ao proposto por [Taniguchi et al. 2005]. As duas abordagens extraem diagramas de sequências com uma fase de simplificação de rastros. A diferença é que no trabalho anterior a fase de compactação é feita sobre os rastros antes de gerar os diagramas enquanto que a fase de compactação proposta por [Delamare et al. 2006] é feita sobre os diagramas de sequência. Neste trabalho, a compressão das repetições é uma etapa de pré-processamento dos rastros. Além de comprimir os rastros, a abordagem proposta sumariza os rastros

comprimidos, baseando na noção de granularidade. Os rastros sumarizados são muito menores que os rastros compactados/comprimidos, pois além de remover as repetições, remove também chamadas de métodos que podem ser irrelevantes para a compreensão. Em adição, o método de sumarização utilizado no presente trabalho não se limita somente a diagramas de sequências, mas contribui também com atividades de evolução de *software*.

A técnica de simplificação dos rastros proposta em [Gargiulo e Mancoridis 2001] é baseada na filtragem de pacotes. A ferramenta lista os nomes dos pacotes selecionados que contêm as classes que serão analisadas. O resultado é apresentado em uma janela, listando as classes que participaram da execução e a dependência entre elas.

Outra abordagem que tem como objetivo sumarizar rastros, [Reiss e Renieris 2001], seleciona subconjuntos de dados de rastros e os compacta para que em um segundo passo os dados sejam codificados na tentativa de inferir sua estrutura. A técnica de seleção é baseada nas classes, a de compactação se baseia nos intervalos de tempo e a codificação é feita em toda a execução como um grafo acíclico direcionado. As técnicas de inferência de estrutura incluem codificação de gramática livre do contexto e a construção de autômato de estados finitos. Na mesma linha de sumarização de padrões, [Systä et al. 2001] apresentam a ferramenta Shimba que usa os diagramas de sequência para visualizar interações entre componentes especificados pelo usuário. A técnica de reconhecimento de padrões é aplicado para lidar com problemas de escalabilidade que está associado com os diagramas. Eles aplicam reconhecimento de padrões para buscar resolver o problema para rastros muito grandes. A técnica aplicada é o algoritmo Boyer-Moore (combinação de *string*) para detectar sequências repetidas de eventos idênticos que se referem como padrões comportamentais. A ferramenta reconhece dois tipos de padrões comportamentais. O primeiro está relacionado com as repetições contínuas, os laços. O segundo consiste de padrões que ocorrem em situações não contínuas no rastro, representando-os em subcenários que são adicionados ao cenário principal. Estratégia semelhante a aplicada neste trabalho, a diferença é que eles omitem as repetições desta maneira, enquanto que neste trabalho os subcenários seriam tarefas importantes do cenário principal e não repetições.

Em [Pauw et al. 1998] é apresentado uma técnica baseada em padrões de execução, onde repetições são comprimidas, semelhante à técnica proposta por [Hamou-lhadj e Lethbridge 2002]. O resultado é apresentado graficamente onde o desenvolvedor pode visualizar as sequências de chamadas de todo um cenário de execução ou visualizar detalhes. A desvantagem desta técnica e de outras apresentadas neste capítulo é que em uma única visão é gerada um cenário inteiro e em sistemas complexos pode se tornar muito difícil a compreensão dos diagramas. A abordagem proposta neste trabalho, além de reduzir os rastros com a aplicação do algoritmo de compressão [Hamou-lhadj e Lethbridge 2002] e do algoritmo *Summarizer*, os rastros são separados por tarefas dos cenários de execução e submetidos a um algoritmo de agrupamento para gerar diagramas menores com o objetivo de simplificar o diagrama principal da tarefa selecionada.

[Dugerdil 2007] propõe uma técnica de redução de rastros de execução baseada em amostras que é aplicada sobre os rastros não comprimidos. O conceito de classe onipresente temporariamente é definido como sendo a analogia de ruídos em processamento de sinal. Durante a análise as classes onipresentes pode ser removidas para concentrar a análise somente nas classes relevantes. Através do uso da técnica de amostras é possível correlacionar os elementos dinamicamente, verificando se eles ocorrem nas mesmas amostras. Com isso as classes podem ser agrupadas dinamicamente para recuperar componentes de sistemas existentes. No estudo de caso, a técnica é aplicada em um sistema relativamente complexo com 240 KLOC no cliente e 90KLOC no servidor.

Em [Cornelissen et al. 2008] é proposto uma metodologia de análise para avaliar e comparar técnicas de redução de rastros de execução. Quatro tipos de métodos de redução encontrados na literatura são apresentados com o objetivo de avaliá-los e compará-los usando um conjunto de sete rastros de execução extremamente volumosos. A abordagem capacita uma análise sistemática de técnicas de redução de rastros que facilita a seleção de reduções diferentes em contextos específicos e permite um uso mais efetivo de ferramentas de análise dinâmica na manutenção de *software*. Os quatro tipos de métodos avaliados são: 1) Técnicas de sumarização tentam diminuir um rastro por substituir parte de seu conteúdo por notações mais concisas se baseando na profundidade da pilha de execução. Alvos de sumarizações típicas incluem padrões de recorrência, ex, notação de padrões de execução [Pauw et al. 1998], sumarização de padrões [Reiss e Renieris 2001], [Systä et al. 2001]. 2) Filtragem baseada em métricas é centrada no uso de certas métricas. Exemplo de tais métricas são profundidade da pilha e graus de *fan-in* e *fan-out*. 3) Técnicas de filtragem baseadas em linguagem buscam omitir chamadas de métodos como *getters* e *setters*, métodos privados. Técnicas como filtragem de pacotes [Gargiulo e Mancoridis 2001] e também *getters* e *setters* [Hamou-Lhadj e Lethbridge 2006]. 4) Abordagens *Ad hoc* estão relacionadas ao uso de técnicas “caixa preta” que não consideram conteúdo do rastro, como técnicas baseadas em amostras [Dugerdil 2007]. Os critérios de avaliação considerados são: taxa de redução bem sucedida, desempenho e preservação de informação. Os testes foram feitos em rastros com pouco mais de 30 mil chamadas até em rastros com mais de 12 milhões de chamadas de métodos com *thresholds* diferentes. Em relação ao critério de avaliação *taxa de redução bem sucedida* o tipo de método que obteve melhor resultado foi o 4º. O 1º e o 2º por serem dependentes da profundidade na pilha de execução, mostraram resultados similares. O 3º tipo não obteve bons resultados, sendo que próximo da metade das reduções não teve sucesso na taxa de redução. Para o critério *desempenho*, o 4º além de obter o melhor resultado na taxa de redução, o obteve em menor tempo. O 2º e o 3º obtiveram os resultados em tempos semelhantes e o 1º foi o que levou mais tempo dos quatro tipos métodos avaliados, porque este tipo geralmente requer que um rastro seja processado várias vezes. Mas em compensação, o 1º tipo de método foi o único que obteve os melhores resultados no critério *preservação de informação*, onde a

porcentagem de eventos de alto nível preservados é significativamente maior que os de nível médio e a porcentagem de nível médio é maior que aqueles eventos de baixo nível. O 2º e 3º obtiveram resultados comparáveis porque nem sempre a porcentagem dos eventos de nível médio mantidos era maior que os de baixo nível. O 4º tipo de método não teve bom resultados, particularmente nos rastros extremamente volumosos. A técnica preservou os três tipos de eventos, alto nível, nível médio e baixo nível, mantendo todos os tipos igualmente representados nos rastros reduzidos. A técnica de sumarização proposta neste trabalho se enquadra no primeiro tipo de método avaliado por [Cornelissen et al. 2008], *sumarização*, fortalecendo nossa hipótese de que chamadas de métodos com granularidade maiores são geralmente as chamadas de métodos mais relevantes para a compreensão do sistema por serem eventos de alto nível.

A diferença da técnica de sumarização utilizada neste trabalho com outras existentes é que informações relevantes são mantidas no rastro, removendo os detalhes de implementação. Os diagramas resultantes por outras abordagens são de uma maneira geral muito grandes e não ajudam muito a compreender o comportamento de sistemas complexos. Rastros de execução de sistemas complexos continuam muito volumosos após a aplicação de métodos de compactação baseados somente nos laços e chamadas recursivas e chamadas de métodos irrelevantes, como eventos de cliques de *mouse*, chamadas de métodos que executam tarefas genéricas, não são removidas.

5.2 Compreensão de Sistemas de *Software*

Várias abordagens para auxiliar na compreensão de sistemas a partir de rastros de execução têm sido propostas como os trabalhos descritos na Seção 5.1. O trabalho apresentado por [Sartipi e Dezhkam 2007] sugere visões dinâmicas e estáticas de um sistema de *software*. A representação da visão dinâmica é através da coleta de informações úteis extraídas da execução de um conjunto de cenários que cobrem as características que são usadas frequentemente. As informações obtidas são embutidas em um processo de recuperação de visão estática. A abordagem une informação estática e dinâmica em uma técnica de recuperação arquitetural baseada em padrões.

[Briand et al. 2006] propuseram uma infraestrutura de metodologia e instrumentação para recuperar diagramas de sequência com o uso da análise dinâmica de sistemas distribuídos em Java e neste trabalho os sistemas estudados são não distribuídos. A instrumentação é a chave da abordagem apresentada em [Briand et al. 2006], eles mostram que a ferramenta não insere no rastro laços e chamadas recursivas. A Ferramenta de instrumentação usada é baseada em AspectJ, semelhante à usada neste trabalho, captura o nome do método qualificado, o nível da chamada na pilha de execução, *timestamp*, identificador do objeto, coleta os parâmetros e recupera valores de retorno das chamadas de métodos, fluxos de controle como `if`, `else`, `while`, `for`, `do`, `switch case` e `return`. No

entanto, a ferramenta de instrumentação neste trabalho não captura valores de retorno e nem fluxos de controle. Os diagramas gerados incluem repetições de mensagens, condições e neste trabalho este tipo de informação não é tratada. A diferença da técnica proposta neste trabalho é que os diagramas podem ser gerados com um nível alto de abstração, por causa da técnica de sumarização e simplificação da *Tree View*, enquanto que a abordagem [Briand et al. 2006] parece recuperar muitos detalhes de implementação. A metodologia apresentada usa metamodelos (como diagrama de classe) e regras de transformações (OCL - Object Constraint Language) para recuperar diagrama de sequências de um cenário de execução alvo, mas o estudo de caso que eles apresentam é um sistema muito simples com somente 7 KLOC. Nos estudos de caso apresentados neste trabalhos um deles foi em um sistema de 163 KLOC. Eles não comentam para casos com sistemas maiores, onde os diagramas de sequência se tornam muito difíceis de entender pelo seu tamanho. O trabalho proposto em [Briand et al. 2003] tem como objetivo definir diagramas de sequências obtidos a partir de rastros de execução. A abordagem se baseia em regras de transformação formais usando (OCL) e diagramas que mostram todas as informações relevantes, incluindo condições, iterações de mensagens e identificadores de objetos específicos. A ferramenta de instrumentação utilizada recupera os mesmos atributos que a apresentada em [Briand et al. 2006]. Primeiro os aspectos relevantes de um diagrama de sequência são modelados como um diagrama de classes (metamodelo) para identificar os requisitos para instrumentação do código fonte. Em seguida modela quais informações os rastros de execução deve conter sob a forma de um metamodelo de rastro (ex: um rastro) para descrever formalmente o mapeamento que deve existir entre os dois metamodelos. A abordagem foi aplicada no mesmo sistema que o estudo de caso apresentado no Capítulo 4 do simulador ATM porém, na versão C++. Este sistema é muito simples, mas viabiliza a avaliação por possuir documentação disponível.

Manutenção de sistemas de software. Para ser feito modificações em um sistema seja ela corretiva, preventiva, adaptativa ou perfectiva, o primeiro passo é entender o comportamento do sistema. Neste trabalho a categoria de manutenção de *software* usada para avaliar a qualidade dos rastros sumarizados é a perfectiva, inserindo novas características semelhantes em um sistema, por aplicar a técnica de alinhamento de rastros de execução proposta por [Paixão 2009] sobre os rastros sumarizados com o objetivo de viabilizar o uso da técnica.

Uma definição que se assemelha a inserção de novas características semelhantes em um sistema é estudada como a prática de fazer cópias de artefatos, ou seja, “clones” [Kim et al. 2004]. O propósito desta ferramenta é que estruturas semelhantes possam orientar os desenvolvedores em suas tarefas de mudança incremental. [Eisenbarth et al. 2003] apresentaram uma técnica semi-automática que reconstrói o mapeamento para características que são disparadas pelo usuário e exibe um comportamento observável. O objetivo da

técnica é identificar partes do código que implementam especialmente uma característica e também o conjunto de código comum e específico para um conjunto de características. Em [Kothari et al. 2007] definem uma medida de sobreposição de implementação de características que determina o quanto similares as características são em suas execuções por analisar seus grafos de chamadas. É considerado o quanto esta medida muda com o tempo e avaliam a hipótese que com o tempo e versões subsequentes de um sistema, as implementações de características similares semanticamente convergem. Como as características de um sistema alvo convergem em suas implementações, grupos de características similares semanticamente são determinados e assim, o custo de compreensão de programas é reduzido por selecionar poucas características chaves que mostram uma visão geral do sistema. O resultado da aplicação da técnica de alinhamento de rastros sumarizados além de detectar elementos, detectam os elementos de código específicos comuns entre características similares semanticamente.

[Greevy et al. 2005] combina análise dinâmica e análise de evolução de múltiplas versões de um sistema. O problema de manipular os rastros muito volumosos é tratado pelas técnicas como, aplicação de medidas para compactar os rastros por reduzir múltiplas referências à mesma classe para um ocorrência e uma medida de histórico que é definida como uma sequência de versões do mesmo tipo de entidade (ex. histórico de classes) para sumarizar mudanças em múltiplas versões. A abordagem de sumarização deste trabalho é diferente por considerar uma chamada de método relevante baseada na sua granularidade e sumarizar os rastros de um único cenário de uma versão do sistema por vez. Enquanto que a sumarização proposta por [Greevy et al. 2005] leva em consideração os rastros de um único cenário de várias versões ao mesmo tempo. A técnica de alinhamento de rastros de execução sumarizados contribui para tarefas de evolução por inserir novas características em um sistema alvo. Enquanto que em [Greevy et al. 2005] o objetivo é detectar as mudanças nos relacionamentos entre classes e características ao longo do tempo.

Em relação a refatoração em código fonte destaca-se os trabalhos de [Weissgerber e Diehl 2006] que propõem um algoritmo que detecta refatorações através de informações contidas em repositórios, depois utiliza de técnicas de detecção de clone para refinar o resultado. Na mesma linha, [Xing e Stroulia 2006] detectam refatorações através de diferentes versões de diagramas UML. [Antoniol et al. 2004] usam técnicas de espaço vetorial para comparar identificadores em diferentes classes para detectar refatorações. Porém, caso haja muitas mudanças nas classes a abordagem não consegue desempenhar sua função muito bem.

[Sillito et al. 2006] propuseram dois estudos qualitativos de desenvolvedores para executar tarefas de evolução de sistemas médios a complexos. O objetivo da análise foi em *quais* informações um desenvolvedor precisa para saber sobre um código enquanto executa uma modificação no sistema e também *como* eles descobrem estas informações. Baseado nesta análise, 44 tipos diferentes de perguntas dos desenvolvedores foram catalogadas e

categorizadas, como também em situações foram questionadas e respondidas. [Sillito et al. 2006] observaram que muitas das perguntas estavam muito relacionadas com algumas questões de baixo nível e sendo parte de respostas de perguntas de alto nível.

[Cornelissen et al. 2009] apresenta uma revisão de literatura sistemática direcionada na identificação e estruturação de pesquisa em compreensão de sistemas através da análise dinâmica. Através de um repositório consistindo de quase cinco mil artigos, 176 artigos foram selecionados e caracterizados em quatro principais observações: atividade, alvo, método e avaliação. Dos vários artigos selecionados e citados em [Cornelissen et al. 2009], são citadas técnicas de compactação de rastros, localização de características, recuperação de diagramas de sequências, mas nenhum dos trabalhos selecionados está relacionado a alinhamento de rastros de execução. Em [Qi et al. 2009] foi proposta uma abordagem automatizada para depuração de programas em evolução utilizando alinhamento de rastros de execução. A abordagem usa dois programas (o programa estável e o programa modificado) e uma entrada teste que executa com sucesso no programa estável mas que falha no programa modificado. Os rastros de ambas as execuções são coletados e então o alinhamento é aplicado sobre os dois rastros. Neste trabalho a técnica de alinhamento também aplica-se sobre dois rastros de execução, porém estes rastros são capturados pela execução de dois cenários de características similares e/ou que tenham o mesmo número de tarefas. Enquanto que o objetivo em [Qi et al. 2009] é localizar *bugs* reais no programa modificado, o objetivo da técnica de alinhamento de rastros de execução sumarizados neste trabalho é localizar elementos específicos e comuns no código fonte para inserir novas características no sistema alvo.

5.3 Considerações Finais

Neste capítulo foram abordados os principais trabalhos relacionados à sumarização de rastros de execução com o objetivo de auxiliar na compreensão de sistemas. Ferramentas que auxiliam na extração de informações de um sistema para auxiliar na inserção de novas características ou na redocumentação. Vários trabalhos correlatos têm os mesmos objetivos aos da abordagem proposta neste trabalho, a simplificar a extração de informação importante dos rastros de execução.

Capítulo 6

Conclusão e Trabalhos Futuros

Este trabalho apresentou uma nova abordagem para auxiliar na compreensão de programas baseada na sumarização de rastros de execução de sistemas complexos que geralmente são muito volumosos, mesmo após a execução de algoritmos para remover repetições. O algoritmo *Summarizer* busca remover as chamadas de métodos menos relevantes assumindo como hipótese que as chamadas de métodos com granularidades maiores são os métodos mais relevantes para a execução de uma característica específica.

Contribuições

Alinhamento de Rastros de Execução Sumarizados Neste trabalho foi apresentada a técnica de alinhamento de rastros de execução sumarizados. Uma visualização gráfica foi proposta para reduzir o esforço de compreensão do resultado do alinhamento entre duas características similares à característica a ser inserida em um sistema. A árvore de alinhamento identifica chamadas de métodos específicas de cada característica por apresentá-las em cores diferentes. Com os rastros sumarizados o número de segmentos desalinhados diminuiu, com isso, viabilizando o uso da técnica. Para avaliar a qualidade dos resultados obtidos foram aplicados três estudos de caso. No primeiro, a ferramenta de modelagem UML ArgoUML, foi implementado o diagrama de objetos. No segundo, o cliente de *email* Columba, foi inserido a característica Envio de Mensagem Agendada. No último, o cliente/servidor de *chat* Llama Chat, a inserção do sintetizador de fala do FreeTTS. Os resultados mostraram uma qualidade expressiva da lista de chamadas métodos selecionados pela abordagem como sendo importantes para inserção de novas características em um sistema que carece de uma boa documentação. Esta abordagem proposta deu origem aos artigos completos em [Silva et al. 2010] e [Silva et al. 2011b].

Extração de Visões de Alto Nível. Neste trabalho também foi apresentada uma técnica semi-automatizada baseada em análise dinâmica, representando os rastros sumarizados em uma *Tree View*. O objetivo desta abordagem é extrair informação importante

para reduzir o esforço de redocumentar sistemas existentes tanto com modelos estruturais quanto comportamentais. Mesmo após a aplicação das técnicas de compressão e sumarização, os rastros de execução ainda permanecem volumosos para recuperar intenções de projeto por análise visual. Esta técnica busca extrair chamadas de métodos relevantes baseada no conceito de granularidade e no nível das mesmas na árvore visual gerada a partir dos rastros sumarizados. Os parâmetros de filtros simplificam o rastro sumarizado representado na *Tree View* sem remover chamadas de métodos do arquivo de rastro. Assim, o analista pode calibrar a visualização da árvore simplificada até obter o resultado próximo do esperado. Para auxiliar na simplificação dos diagramas de sequência, um agrupamento é feito em modo *bottom-up* na árvore que sugerem possíveis componentes, por colorir as sub-árvores selecionadas. A abordagem não requer uma configuração complexa e a execução é simples e razoavelmente rápida em qualquer computador de *prateleira*. Dois estudos de caso foram apresentados com o objetivo de avaliar a qualidade dos rastros sumarizados, fazendo uma comparação com a documentação disponível dos sistemas selecionados. Uma visualização gráfica simplificada dos rastros sumarizados foi gerada para facilitar a extração de informações com alto nível de abstração. Os resultados de precisão e *recall* foram promissores levando em consideração o espaço de busca, especialmente rastros muito volumosos como os do Tomcat. Obviamente para considerar a qualidade dos resultados em termos de correção, nem todos os métodos recuperados pela técnica precisam estar contidos na documentação. Esta técnica proposta deu origem ao artigo completo em [Silva et al. 2011a].

Sumarização de Rastros. Neste trabalho foi apresentado um novo conceito para explorar rastros de execução baseado na sumarização dos rastros extremamente volumosos. O objetivo do algoritmo de sumarização é remover chamadas de métodos de baixo nível que não influenciam na compreensão, mas que dificultam a extração de chamadas de métodos específicas de uma determinada característica de um sistema. A abordagem de sumarização é baseada na filtragem de chamadas com baixa granularidade, buscando remover detalhes de implementação que pode ser chamadas como eventos de *mouse* e chamadas de métodos que não são específicas de uma determinada característica. A abordagem foi avaliada por executar duas técnicas sobre os rastros sumarizados. Os resultados obtidos, tanto da técnica de alinhamento quanto a técnica de extração de visões de alto nível, mostraram que a abordagem de sumarização ajuda a compreender as dinâmicas de um sistema com documentação deficiente.

Limitações

Embora os resultados obtidos da aplicação das duas técnicas sobre os rastros sumarizados sejam muito promissores, chamadas de métodos que são utilitários ou ruídos não

são todos removidos. Os resultados da aplicação do alinhamento de rastros obteve uma precisão máxima de 80% das chamadas de métodos apontadas como relevantes. Este resultado foi devido ao razoável número de falsos positivos que são métodos que a abordagem apontou como chamadas de métodos importantes mas que deveriam ter sido removidas durante a sumarização. A limitação da técnica que teve como objetivo extrair visões de alto nível teve o *recall* muito baixo devido ao número de falsos negativos que são as chamadas de métodos que deveriam estar presentes no rastro sumarizado mas são removidas pelo algoritmo de sumarização. Algumas das limitações que foram identificadas neste trabalho são:

Padrões de Repetição. Mesmo o processo de sumarização consistindo de três fases, onde o algoritmo de compressão é executado duas vezes, muitos padrões de repetição não são detectados.

Chamadas de Métodos Pertencentes a Utilitários. Mesmo após a sumarização dos rastros, várias chamadas de métodos que não são específicas de uma característica permanecem nos rastros sumarizados.

Remoção de Chamadas de Métodos Específicas. Alguns métodos que pertencem a uma determinada característica são removidas pelo algoritmo de sumarização, aumentando os falsos negativos durante a avaliação de qualidade dos resultados.

Dependência de Análise do Usuário. Atualmente, a identificação das chamadas de métodos relevantes/irrelevantes depende da percepção do analista.

Trabalhos Futuros

Alguns trabalhos futuros podem ser realizados para melhorar os resultados da sumarização dos rastros, como também aplicar a abordagem em novos contextos:

Melhoria do Algoritmo de Sumarização. Melhorar o algoritmo de sumarização para gerar um rastro sumarizado com maior qualidade. Os itens a seguir descrevem objetivos específicos:

1. **Padrões de Repetição.** Uma possível solução para este problema seria aplicar mineração de padrões sequenciais.
2. **Chamadas de Métodos Pertencentes a Utilitários.** Com o objetivo de aumentar a precisão das chamadas de métodos apontadas pela abordagem, uma possível solução para este problema seria detectar as chamadas de métodos que estão presentes em diferentes cenários de execução.

3. **Remoção de Chamadas de Métodos Específicas.** Verificar se é possível detectar os problemas que acarretam esta remoção e elaborar uma solução para eliminá-los do algoritmo atual.
4. **Dependência de Análise do Usuário.** Com a solução dos problemas apresentados é esperado que o resultado do processo de seleção das chamadas de métodos não dependa sensivelmente de análise subjetiva do usuário.

Aplicação do Algoritmo de Sumarização em Novos Contextos. Ampliar o contexto de aplicação do algoritmo de sumarização após melhoria, para problemas como:

1. **Detecção de interesses transversais e/ou padrões de projeto.** Poucos testes foram feitos ainda com os rastros sumarizados. A aplicação de uma técnica para mineração de padrões, com o objetivo de detectar interesses transversais, detectou alguns métodos do sistema HealthWatcher¹ que foram apontados pela abordagem e que estão relacionados a padrões, conforme verificação baseada na documentação do sistema.
2. **Detecção de padrões arquiteturais.** Elaborar uma abordagem de recuperação de arquitetura por meio da detecção de padrões. Uma maneira eficiente para especificar padrões deve ser definida para que possam ser encontrados padrões na árvore sumarizada gerada pela técnica apresentada no Capítulo 4, assim verificando se uma arquitetura segue um determinado padrão.

¹<http://www.comp.lancs.ac.uk/greenwop/tao/implementation.htm>

Referências Bibliográficas

- [Antoniol et al. 2004] Antoniol, G., Penta, M. D., e Merlo, E. (2004). An Automatic Approach to Identify Class Evolution Discontinuities. In *7th International Workshop on Principles of Software Evolution*, IWPSE 2004, pp. 31–40, Kyoto, Japan. IEEE Computer Society.
- [Ball 1999] Ball, T. (1999). The Concept of Dynamic Analysis. In *Proceedings of the 7th European Software Engineering Conference*, pp. 216–234. Springer-Verlag.
- [Bennett e Rajlich 2000] Bennett, K. e Rajlich, V. T. (2000). Software Maintenance and Evolution: a Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pp. 73–87, Limerick, Ireland. ACM.
- [Briand et al. 2006] Briand, L., Labiche, Y., e Leduc, J. (2006). Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering*, 32:642–663.
- [Briand et al. 2003] Briand, L., Labiche, Y., e Miao, Y. (2003). Towards the Reverse Engineering of UML Sequence Diagrams. In *10th International Working Conference on Reverse Engineering*, WCRE 2003.
- [Clements et al. 2002] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., e Stafford, J. (2002). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley.
- [Corbi 1989] Corbi, T. (1989). Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306.
- [Cornelissen et al. 2008] Cornelissen, B., Moonen, L., e Zaidman, A. (2008). An Assessment Methodology for Trace Reduction Techniques. In *24th IEEE International Conference on Software Maintenance*, ICSM 2008, Beijing.
- [Cornelissen et al. 2009] Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., e Koschke, R. (2009). A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35:684–702.
- [Delamare et al. 2006] Delamare, R., Baudry, B., e Traon, Y. L. (2006). Reverse-engineering of UML 2.0 Sequence Diagrams from Execution Traces. In *International Workshop on Object-Oriented Reengineering*, WOOR 2006.
- [Ducasse e Pollet 2009] Ducasse, S. e Pollet, D. (2009). Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591.

- [Dugerdil 2007] Dugerdil, P. (2007). Using Trace Sampling Techniques to Identify Dynamic Clusters of Classes. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '07, pp. 306–314, Richmond Hill, Ontario, Canada. ACM.
- [Eisenbarth et al. 2003] Eisenbarth, T., Koschke, R., e Simon, D. (2003). Locating Features in Source Code. *IEEE Transactions on Software Engineering*, 29:210–224.
- [Gargiulo e Mancoridis 2001] Gargiulo, J. e Mancoridis, S. (2001). Gadget: A Tool for Extracting the Dynamic Structure of Java Programs. In *Proceedings 13th International Conference on Software Engineering and Knowledge Engineering*, SEKE 2001, Buenos Aires, Argentina. IEEE Computer Society.
- [Greevy et al. 2005] Greevy, O., Ducasse, S., e Girba, T. (2005). Analyzing Feature Traces to Incorporate the Semantics of Change in Software Evolution Analysis. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM 2005, pp. 347–356, Washington, DC, USA. IEEE Computer Society.
- [Gusfield 1997] Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press.
- [Hamou-Lhadj et al. 2005] Hamou-Lhadj, A., Braun, E., Amyot, D., e Lethbridge, T. (2005). Recovering Behavioral Design Models from Execution Traces. In *the 9th European Conference on Software Maintenance and Reengineering*, CSMR 2005, pp. 112–121, Manchester, UK. IEEE Computer Society.
- [Hamou-lhadj e Lethbridge 2002] Hamou-lhadj, A. e Lethbridge, T. (2002). Compression Techniques to Simplify the Analysis of Large Execution Traces. In *10th International Workshop on Program Comprehension*, IWPC 2002, pp. 159–168, Washington, DC, USA. IEEE Computer Society.
- [Hamou-lhadj e Lethbridge 2004] Hamou-lhadj, A. e Lethbridge, T. (2004). Reasoning About the Concept of Utilities. In *1st International Workshop on Practical Problems of Programming in the Large*, ECOOP 2004, pp. 10–22, Oslo, Norway. Springer-Verlag.
- [Hamou-Lhadj e Lethbridge 2006] Hamou-Lhadj, A. e Lethbridge, T. (2006). Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC 2006, pp. 181–190, Washington, DC, USA. IEEE Computer Society.
- [Jacobson 1995] Jacobson, I. (1995). *Object-Oriented Software Engineering: a Use Case driven Approach*. Addison-Wesley.
- [Jerding e Rugaber 1997] Jerding, D. e Rugaber, S. (1997). Using Visualization for Architectural Localization and Extraction. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, WCRE 1997.
- [Kim et al. 2004] Kim, M., Bergman, L., Lau, T., e Notkin, D. (2004). An Ethnographic Study of Copy and Past Programming Practices in oopl. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, ISESE 2004, pp. 83–92, Washington, DC, USA. IEEE Computer Society.

- [Kothari et al. 2007] Kothari, J., Denton, T., Shokoufandeh, A., e Mancoridis, S. (2007). Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC 2007, pp. 17–26, Banff, Alberta, Canada. IEEE Computer Society.
- [Kuhn e Greevy 2006] Kuhn, A. e Greevy, O. (2006). Summarizing Traces as Signals in Time. In *Workshop on Program Comprehension through Dynamic Analysis*, PCODA 2006, pp. 1–6, Benevento, Italy. IEEE Computer Society.
- [Lethbridge et al. 2003] Lethbridge, T. C., Singer, J., , e Forward, A. (2003). How Software Engineers use Documentation: The State of the Practice. 20(6):35–39.
- [Lucia et al. 2009] Lucia, A. D., Oliveto, R., e Tortora, G. (2009). Assessing IR-based Traceability Recovery Tools through Controlled Experiments. *Empirical Software Engineering*, 14:57–92.
- [Maia et al. 2008] Maia, M., Sobreira, V., Paixão, K., e Amo, S. (2008). Using a Sequence Alignment Algorithm to Identify Specific and Common Code from Execution Traces. In *4th International Workshop on Program Comprehension through Dynamic Analysis*, pp. 6–11.
- [Mit e Ernst. 2003] Mit, M. e Ernst., M. (2003). Static and Dynamic Analysis: Synergy and Duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pp. 24–27.
- [Müller et al. 1993] Müller, H., Orgun, M., Tilley, S., e Uhl, J. (1993). A Reverse-Engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, pp. 181–204.
- [Myers et al. 2010] Myers, D., Storey, M., e Salois, M. (2010). Utilizing Debug Information to Compact Loops in Large Execution Traces. In *the 14th European Conference on Software Maintenance and Reengineering*, CSMR 2010, pp. 41–50, Madrid, Spain. IEEE Computer Society.
- [Needleman e Wunsch 1970] Needleman, S. B. e Wunsch, C. D. (1970). A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, (48):443–453.
- [Paixão 2009] Paixão, K. V. R. (2009). Alinhamento de Rastros de Execução de Programas para Compreensão de Pontos de Variação em Código Fonte. In *Master's Thesis*, Uberlândia, MG, Brasil.
- [Pauw et al. 1998] Pauw, W. D., Lorenz, D., Vlissides, J., e Wegman, M. (1998). Execution Patterns in Object-Oriented Visualization. In *Proceedings 4th USENIX Conference on Object-Oriented Technologies and Systems*, COOTS 1998. USENIX.
- [Pennington 1987] Pennington, N. (1987). Comprehension Strategies in Programming. In *Second Workshop on Empirical Studies of Programmers*, pp. 100–113. Ablex Publishing Corporation.

- [Prada-Rojas et al. 2010] Prada-Rojas, C., Santana, M., De-Paoli, S., e Raynaud, X. (2010). Summarizing Embedded Execution Traces Through a Compact View. In *Conference on System Software, SoC and Silicon Debug, S4D 2010*, pp. 41–50, Southampton, UK. IEEE Computer Society.
- [Qi et al. 2009] Qi, D., Roychoudhury, A., Liang, Z., e Vaswani, K. (2009). Darwin: an Approach for Debugging Evolving Programs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09*, pp. 33–42, Amsterdam, The Netherlands. ACM.
- [Reiss e Renieris 2001] Reiss, S. e Renieris, M. (2001). Encoding Program Executions. In *Proceedings 23rd International Conference on Software Engineering, ICSE 2001*. ACM.
- [Sartipi e Dezhkam 2007] Sartipi, K. e Dezhkam, N. (2007). An amalgamated Dynamic and Static Architecture Reconstruction Framework to Control Component Interactions. In *Proceedings of the 14th IEEE Working Conference on Reverse Engineering, WCRE 2007*, pp. 259–268, Vancouver, Canada.
- [Seacord et al. 2003] Seacord, R., Plakosh, D., e Lewis, G. (2003). *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley.
- [Sillito et al. 2006] Sillito, J., Murphy, G., e Volder, K. D. (2006). Questions Programmers Ask During Software Evolution Tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14*, pp. 23–34, Portland, Oregon, USA. ACM.
- [Silva et al. 2011a] Silva, L., Amo, S., e Maia, M. (2011a). Sumarização de Rastros de Execução para Recuperação de Visões de Alto Nível em Sistemas Orientados a Objetos. In *14th Ibero-American Conference on Software Engineering, Cibse 2011*.
- [Silva et al. 2010] Silva, L., Paixão, K., Amo, S., e Maia, M. (2010). Software Evolution Aided by Execution Trace Alignment. In *24th Brazilian Symposium on Software Engineering, SBES 2010*, pp. 158–167, Los Alamitos, CA, USA. IEEE Computer Society.
- [Silva et al. 2011b] Silva, L., Paixão, K., Amo, S., e Maia, M. (2011b). On the Use of Execution Trace Alignment for Driving Perfective Changes. In *15th European Conference on Software Maintenance and Reengineering, CSMR 2011*.
- [Sobreira e Maia 2008] Sobreira, V. e Maia, M. (2008). A Visual Trace Analysis Tool for Understanding Feature Scattering. In *Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008*.
- [Soloway et al. 1988] Soloway, E., Adelson, B., e Ehrlich, K. (1988). Knowledge and Processes in the Comprehension of Computer Programs. *The Nature of Expertise*, pp. 239–152.
- [Soloway e Ehrlich 1984] Soloway, E. e Ehrlich, K. (September 1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609.

- [Subcommittee 1992] Subcommittee, S. E. S. (1992). *IEEE Standard for Software Maintenance*. IEEE Std 1219-1993. IEEE Computer Society.
- [Systä et al. 2001] Systä, T., Koskimies, K., e Müller, H. (2001). Shimba, an Environment for Reverse Engineering Java Software Systems. *Software - Practice and Experience*, 31:371–394.
- [Taniguchi et al. 2005] Taniguchi, K., Ishio, T., Kamiya, T., Kusumoto, S., e Inoue, K. (2005). Extracting Sequence Diagram from Execution Trace of Java Program. In *the 8th International Workshop on Principles of Software Evolution, IWPSE 2005*, pp. 148–154, Lisbon, Portugal.
- [von Mayrhauser e Vans 1995] von Mayrhauser, A. e Vans, A. M. (1995). Program Comprehension During Software Maintenance and Evolution. *Computer*, 28:44–55.
- [Weissgerber e Diehl 2006] Weissgerber, P. e Diehl, S. (2006). Identifying Refactorings from Source-Code Changes. In *the 21st International Conference on Automated Software Engineering, ASE 2006*, pp. 231–240, Washington, DC, USA. IEEE Computer Society.
- [Xing e Stroulia 2006] Xing, Z. e Stroulia, E. (2006). Refactoring Detection Based on UmlDiff Change-Facts Queries. In *Proceedings of the 13th IEEE Working Conference on Reverse Engineering, WCRE 2006*, pp. 263–274. IEEE Computer Society.
- [Yan et al. 2004] Yan, H., Garlan, D., Schmerl, B., Aldrich, J., e Kazman, R. (2004). DiscoTect: A System for Discovering Architectures from Running Systems. In *Proceedings 26th International Conference on Software Engineering, ICSE 2004*, pp. 470–479.
- [Zaidman 2006] Zaidman, A. (2006). *Scalability Solutions for Program Comprehension through Dynamic Analysis*. PhD thesis, University of Antwerp.