Software Evolution Aided by Execution Trace Alignment[1]

Luciana Lourdes Silva, Klérisson Ribeiro Paixão, Sandra de Amo, Marcelo de Almeida Maia
Computer Science Department
Federal University of Uberlândia
Uberlândia, Brazil
luciana.lourdes@gmail.com, deamo@ufu.br, marcmaia@facom.ufu.br

***Resumo* — Várias alternativas para facilitar o entendimento do comportamento de sistemas de software têm sido propostas. Contudo, ainda não existe uma abordagem amplamente aceita para facilitar o entendimento de sistemas com código-fonte disponível e baixo grau de documentação com o objetivo de permitir que novos desenvolvedores possam contribuir com a evolução desta classe de sistemas. O grau de dificuldade para entender sistemas complexos pode tornar a tarefa de evolução impeditiva para muitos destes desenvolvedores. Este trabalho propõe um novo método baseado na análise de rastros de execução para localizar os pontos no código fonte onde devem ser executadas as alterações para introduzir novas funcionalidades. O método foi avaliado com um estudo em um sistema real, onde foram inseridas novas funcionalidades com um baixo esforço.**

***Abstract* — Several attempts to facilitate understanding the behavior of software systems have been proposed. Nonetheless, there is no widely accepted approach to facilitate understanding software systems with poor documentation with the goal that new developers could contribute with the evolution of these systems. The effort to understand complex systems may be a prohibitive factor in program comprehension tasks for many developers. This work proposes a new method based on the analysis of execution traces for locating points in the source code where changes that introduce new functionalities should be performed. The proposed method was evaluated with a real world system, where new functionalities were inserted with low effort.**

Palavras chave – evolução de software; rastros de execução; reuso; manutenção de software

Keywords – software evolution; execution traces; reuse; software maintenance

I. INTRODUÇÃO

A quantidade de código fonte aberto disponível na Internet é grande o suficiente para prover código útil que possa ser incorporado aos produtos finais pela grande maioria das equipes de desenvolvimento. Entretanto, é reconhecido que a maior parte deste software carece de uma documentação adequada ou quando esta documentação existe, não há garantias que esteja atualizada ou completa [7].

A maior parte do esforço das atividades de modificação de um produto de software para adicionar novas funcionalidades decorre do entendimento de requisitos e artefatos de código. Um dos problemas mais significativos nas atuais práticas de evolução de software é a dificuldade dos mantenedores encontrarem seções nos códigos fontes onde precisam ocorrer alterações ou onde novas características precisam ser inseridas, demonstrando que a rastreabilidade entre requisitos e artefatos de código ainda é um grande desafio para os desenvolvedores [2].

Várias tentativas para facilitar o entendimento de sistemas de software têm sido propostas baseadas na hipótese de que o próprio código fonte é a única fonte confiável de informação sobre o software disponível [14]. Contudo, as técnicas atuais para compreensão de programas são insuficientes para possibilitar o rápido entendimento da implementação de uma funcionalidade. Os ambientes de desenvolvimento recentes provêem ferramentas poderosas de depuração para compreender o comportamento do sistema, mas provêem somente visões simplificadas para localizar mais facilmente os pontos onde devem ocorrer a implementação de novas funcionalidades no código fonte.

Este trabalho pretende contribuir no seguinte cenário. *Suponha a existência de um sistema de código fonte disponível, o qual o desenvolvedor não tenha participado do processo de desenvolvimento e nem tenha conhecimento prévio do mesmo, e ainda para o qual não se tenha documentação da implementação, ou não se deseja usar extensivamente a documentação existente. Suponha que o objetivo seja incluir uma nova funcionalidade no sistema que seja similar a alguma outra já existente. Como um exemplo deste cenário, suponha que um desenvolvedor deseje evoluir uma ferramenta de edição de diagramas UML e que este desenvolvedor não tenha tido nenhum contato com a implementação da mesma. Suponha que seja desejado incluir novos diagramas e que se assume como hipótese que os novos diagramas tenham implementação similar às implementações dos diagramas existentes*.

Atualmente, as abordagens tradicionais para efetuar esta evolução incluem como parte das suas tarefas, o entendimento do sistema por meio da análise da documentação existente e/ou a depuração das funcionalidades similares como uma alternativa para entendimento inicial do sistema. Uma das hipóteses deste trabalho é que, analisando artefatos semelhantes existentes em um sistema, a inserção de uma nova funcionalidade semelhante a alguma existente possa ter seu tempo e esforço reduzido, supondo que é possível descobrir onde serão inseridos novos artefatos ou alterados artefatos existentes e até como será a forma inicial dos artefatos.

[1] Evolução de Sistemas Apoiada por Alinhamento de Rastros de Execução

978-0-7695-4273-7/10 $26.00 © 2010 IEEE
DOI 10.1109/SBES.2010.25

A presente proposta é fundamentada no uso de técnicas de alinhamento de sequências sumarizadas de rastros de execução, com o objetivo de separar código comum de código específico entre as sequências, identificando as variações com precisão. As sequências são rastros de execução de software definidas por roteiros de execução similares. A principal contribuição deste trabalho está em localizar potenciais elementos de código que podem direcionar o desenvolvimento de uma nova funcionalidade, bem como a sequência de eventos que deverão ocorrer para que a nova funcionalidade seja incorporada. No estudo de caso para avaliar a técnica proposta foi utilizada a ferramenta de modelagem ArgoUML [15].

O artigo é organizado como se segue. A Seção II apresenta a visão geral da proposta. É apresentada a forma de coleta, representação e sumarização dos rastros de execução. São apresentados o funcionamento do alinhamento de rastros e como é usado o resultado deste alinhamento. A Seção III relata o estudo de caso realizado. A Seção IV apresenta a discussão sobre os resultados encontrados. A Seção V apresenta trabalhos relacionados. A última seção apresenta a conclusão e trabalhos futuros.

II. A ABORDAGEM PROPOSTA

O objetivo deste trabalho é extrair informação do código fonte que direcione a compreensão de sistemas de forma a facilitar as tarefas de evolução de software. Neste sentido será utilizada uma técnica para alinhamento de rastros de execução de um programa alvo, no intuito de encontrar pontos comuns e de variação na implementação de cenários de execução semelhantes. Assim, a técnica é aplicável a um caso em particular: dadas características semelhantes em um sistema, o problema é encontrar qual é a localização dos potenciais pontos de modificação na implementação para se inserir uma nova característica e qual é a estrutura desta modificação.

A abordagem é composta basicamente por 4 fases:

A) Definição e Execução de Cenários de Execução,

B) Compressão e Sumarização de Rastros de Execução,

C) Alinhamento de Rastros de Execução e

D) Análise Visual da Árvore de Alinhamento com Implementação das Novas Funcionalidades.

A primeira fase requer a coleta dos rastros de execução das funcionalidades similares que servirão de base para a implementação da nova funcionalidade. O desenvolvedor define quais cenários de execução são interessantes para a análise e manutenção a ser realizada. Estes cenários devem ser expressos em termos das funcionalidades similares. O sistema alvo deve ser recompilado com suporte a aspectos para a coleta do rastro por nossa ferramenta de instrumentação, baseada em AspectJ. Imediatamente antes da execução de um cenário, deve-se informar à ferramenta de instrumentação que os eventos (chamadas de métodos) a partir daquele ponto estão relacionados a uma funcionalidade específica. A coleta do rastro produz como resultado um arquivo com a sequência de chamada de métodos para cada *thread* iniciada pelo sistema alvo. Cada linha do arquivo corresponde a uma chamada de método devidamente qualificado e anotado com a profundidade deste método na pilha de execução.

O módulo de alinhamento de rastros é responsável por produzir uma visualização que ajudará na compreensão desejada. A entrada para o alinhamento são duas sequências de tamanhos potencialmente diferentes. Ao final do alinhamento, são produzidas duas novas sequências de mesmo tamanho, com espaços inseridos nas mesmas de tal forma que métodos alinhados aparecerão na mesma posição nas duas sequências.

A hipótese deste trabalho é que os pontos de desalinhamento nas sequências são potenciais trechos de implementação das funcionalidades que são diferentes entre si e que, portanto indicam o que deve ser implementado ou modificado para suportar a nova funcionalidade a ser introduzida. Entretanto, os desalinhamentos também podem representar algum ruído do rastro.

Além disso, é possível visualizar as sequências de métodos que foram executados anteriormente e posteriormente aos métodos que variaram. Tais sequências servem de base para as análises de impacto que deverão ser feitas para completar o processo de mudança incremental.

A seguir detalharemos mais as 4 fases da abordagem.

A. *Definição e Execução de Cenários*

A escolha dos cenários é baseada em funcionalidades que se assemelham à nova funcionalidade a ser inserida. Inicialmente, são definidos dois cenários semelhantes com um número pequeno de subfuncionalidades, como por exemplo, a inicialização do sistema, pois uma análise sobre rastros com poucas subfuncionalidades tende a ser bem mais simples. Uma subfuncionalidade é definida como sendo uma tarefa que precisa ser executada em uma determinada funcionalidade. Para definir cenários relevantes, o desenvolvedor precisa analisar as funcionalidades existentes no sistema com o objetivo de detectar quais são as que mais se assemelham com a nova funcionalidade a ser

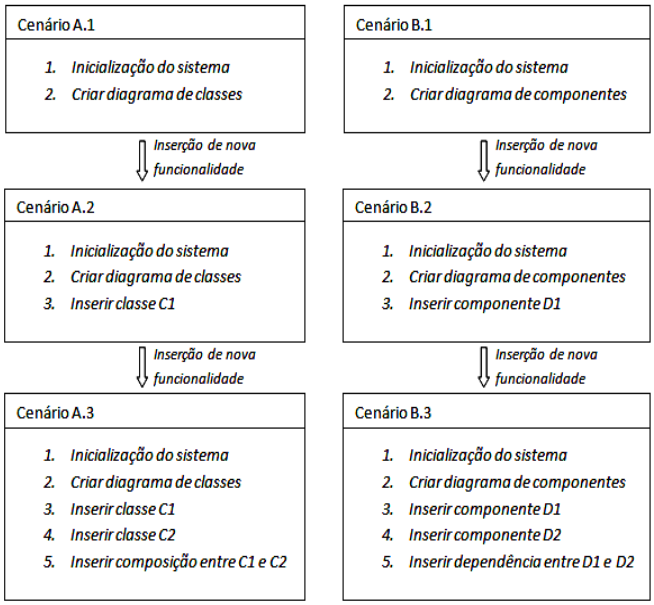

Figura 1. Cenários utilizados para alinhamento.

implementada, ou seja, quais funcionalidades possuem subfuncionalidades semelhantes como mostra na Figura 1. Depois de detectado os cenários semelhantes, os dois cenários que mais se assemelham com a nova funcionalidade são selecionados. Se caso houver subfuncionalidades iguais em um dos cenários selecionados com o da nova funcionalidade é possível detectar após o alinhamento como reutilizá-las.

O processo de seleção dos cenários se resume em dois tipos:

1) Incremental: quando as subfuncionalidades são dependentes entre si. Por exemplo: Para inserir a funcionalidade diagrama de objetos no ArgoUML foram definidos dois tipos de cenários de execução, ver Figura 1, onde para cada tipo foram criados três sub-cenários. No primeiro passo foram utilizadas somente as subfuncionalidades *Inicialização do sistema* e *criar diagrama de classes/componentes*. No segundo passo, para o novo cenário foram utilizadas as subfuncionalidades do cenário anterior mais uma nova subfuncionalidade. O processo se repetiu até obter o cenário final que contenha todas as subfuncionalidades semelhantes com as da nova funcionalidade diminuindo o esforço de implementação. Esta estratégia tem como objetivo simplificar a análise visual após o alinhamento, devido ao grande volume de chamadas de métodos nos rastros de execução. A cada passo que é adicionado uma subfuncionalidade em cada um dos dois cenários selecionados, o desenvolvedor já estará familiarizado com o cenário anterior, daí facilitando de uma maneira direta a localização dos métodos utilizados para executar a funcionalidade adicionada.

2) Independentes: é quando as subfuncionalidades são independentes. Este caso é mais simples pois haverá somente duas subfuncionalidades por cenário, o de *inicialização do sistema* e a subfuncionalidade a ser avaliada. Então para cada tipo de cenário, o novo cenário se diferenciará somente da subfuncionalidade a ser inserida, mantendo somente a(s) subfuncionalidade(s) básicas.

Através da execução de funcionalidades existentes no sistema é possível o desenvolvedor detectar se as subfuncionalidades executadas são dependentes ou independentes. Por exemplo: Para inserir uma associação em um diagrama de classes, é necessário criar pelo menos duas classes no diagrama, isto é, associação é uma subfuncionalidade dependente. Observe que na Figura 1 a subfuncionalidade *inicialização do sistema* é utilizada, que faz parte do tipo incremental, e que na definição do tipo independentes é citado como sendo uma subfuncionalidade presente nos cenários. Esta funcionalidade é importante estar presente nos cenários porque nela pode estar a inicialização das funcionalidades existentes no sistema, a adição de itens de *menu*, habilitação/adição de botões em barras de ferramentas no painel principal que fazem ligação às funcionalidades existentes no sistema. Por exemplo: Com análise desta subfuncionalidade no ArgoUML foi possível detectar onde é criado os itens de menu dos diagramas, das críticas, atalhos para criar um diagrama, salvar, etc.

Após os cenários serem definidos, o sistema é então instrumentado para coletar rastros durante a execução de um cenário específico. Um arquivo de rastro de execução é gerado para cada *thread* iniciada durante a execução do programa. Cada linha de um arquivo de rastro de execução contém informações sobre a chamada de método: o nome do pacote, da classe e do método, uma representação textual dos valores de parâmetros, o tempo quando a execução do método iniciou e o nível da pilha de execução em que o método foi chamado. O nível da chamada informa qual método chamou o método atual e com isto é possível construir uma árvore de chamadas que representa o rastros de execução.

B. Compressão e Sumarização de Rastros de Execução

Os rastros de execução de programas são sequências de chamadas de métodos, as quais podem conter centenas de milhares, e até milhões de chamadas. Uma alternativa para diminuir o tamanho dos rastros de execução é a eliminação de chamadas dentro de repetições e chamadas recursivas [4]. Entretanto, mesmo após a compressão do rastro, foi mostrado em resultados preliminares que o resultado de alinhamento de rastros de execução pode resultar em uma quantidade de informação extremamente grande, que pode inviabilizar a utilidade da abordagem [8,11].

Neste trabalho, propomos além da compressão do rastro uma técnica de sumarização baseada na eliminação de chamadas de menor grau de importância. A definição do grau de importância de uma chamada de método é baseada na quantidade total de outras chamadas que foram feitas internamente à chamada em questão. Este valor é denominado **granularidade** da chamada. Ou seja, quanto maior o número de chamadas internas (granularidade) a uma determinada chamada, maior é a importância relativa desta chamada em relação às outras. A granularidade de sumarização a ser utilizada na abordagem deve ser definida pelo usuário de maneira a balancear a relação precisão / abrangência da árvore obtida. Quanto menor a granularidade escolhida menor é a precisão, pois uma granularidade pequena indica que métodos com poucas chamadas internas são considerados, o que aumenta o número de métodos a serem analisados. Por outro lado, uma granularidade pequena favorece a cobertura, pois descarta menos chamadas de métodos.

C. Alinhamento de Rastros de Execução

Uma importante contribuição para o problema de alinhamento de sequências foi proposto por Needleman e Wunsch [10]. Trata-se de um algoritmo de programação dinâmica para comparar sequências de forma global, ou seja, considerando toda a sequência. Contudo, alguns aspectos devem ser considerados na adaptação deste algoritmo para os propósitos da manutenção de software.

A primeira adaptação refere-se às características de um rastro de execução. Os rastros de execução podem apresentar ruídos. Por exemplo, nas sequências ``XaaaaY'' e ``XaaaY'',

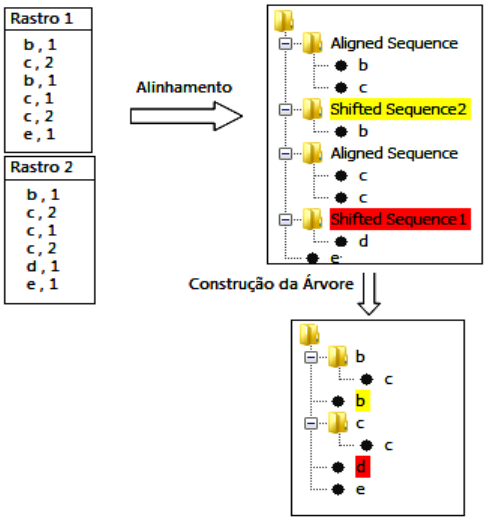

Figura 2. Resultado de alinhamento de sequências.

pode-se suspeitar que tais sequências foram geradas pelo mesmo código. A diferença pode ter sido causada por um loop que executou o método "a" por 4 vezes na primeira sequência e 3 na outra, e por isto a importância de se comprimir os rastros.

Na literatura são comumente encontrados dois tipos de abordagem: Alinhamento Local e Alinhamento Global [3]. O resultado que se deseja obter é preponderante na escolha do tipo de alinhamento. Se o objetivo do alinhamento é comparar sequências com um alto grau de similaridade e com tamanhos semelhantes, o alinhamento global é o mais indicado. O alinhamento de sequências do tipo local é mais utilizado para os casos onde o propósito do alinhamento é encontrar uma sequência menor dentro de uma sequência maior. Neste trabalho, o tipo de alinhamento escolhido foi o alinhamento global, pois será utilizado para comparar sequências que possuem um alto grau de similaridade e com tamanhos semelhantes, onde o objetivo é encontrar características semelhantes e pontos de variabilidades nas sequências.

Uma segunda possibilidade sobre algoritmos de alinhamento está relacionada à quantidade de sequências que serão alinhadas simultaneamente. O alinhamento de pares de sequências é utilizado em um alinhamento global ou local, de duas sequências. No alinhamento de várias sequências é necessário fazer uma integração de técnicas já que a comparação só pode ser feita com duas sequências por vez. Os algoritmos de alinhamento de múltiplas sequências são mais indicados para sequências pequenas, pois este problema é NP-completo. Como neste trabalho as sequências que serão comparadas possuem um tamanho grande, o alinhamento de pares foi escolhido.

Por fim, uma terceira possibilidade para algoritmos de alinhamento diz respeito à decisão se as sequências serão alinhadas considerando políticas de substituição entre os elementos ou não. No primeiro caso uma matriz de substituição deve ser definida e as pontuações dos alinhamentos são calculadas levando em consideração os valores presentes na matriz. Esse esquema de alinhamento é pertinente quando se conhece as relações entre os elementos em análise e pode-se quantificar o quanto um elemento é semelhante a outro. Dessa forma esse tipo de abordagem permite que elementos diferentes na sequência sejam alinhados. Por outro lado, o alinhamento pode ser feito somente entre elementos idênticos, ou seja, se um elemento em análise é diferente de outro ocorre um desalinhamento. Neste trabalho, optou-se pelo alinhamento de elementos idênticos.

Os dois rastros podem ser quebrados em segmentos, os quais estão na mesma posição em cada rastro. As segmentações dos rastros são determinadas agrupando elementos alinhados em sequência e agrupando também os elementos desalinhados em sequência em cada rastro.

A apresentação do resultado do alinhamento entre o par de rastros é dada em uma árvore, pois é considerado o nível de chamadas de métodos na pilha de execução e assim é possível uma análise mais amigável. Os segmentos desalinhados são coloridos, cujos nós têm a mesma coloração que (*Shifted Sequence 2*), se o segmento desalinhado contém os métodos do primeiro rastro, e (*Shifted Sequence 1*) se contém métodos do segundo rastro. A Figura 2 apresenta um exemplo de alinhamento de rastro. São apresentados dois rastros, contendo o nome dos métodos chamados (*b,c,d,e*) e seu respectivo nível na pilha de execução. A sequência <*(b,1), (c,2)*> indica que o método *c* foi chamado dentro do método *b*. Na árvore final, o primeiro segmento apresenta as chamadas *b* e *c*, onde a chamada a *c* é filha da chamada a *b*, por causa do nível de chamada. O segundo segmento contém um desalinhamento (*Shifted Sequence 2*), indicando que a chamada ao método *b* no rastro 1 não tem contraparte no rastro 2. Em seguida, tem-se um segmento de alinhamento contendo duas chamadas (recursivas) ao método *c*. Depois tem-se um segmento de desalinhamento (*Shifted Sequence 1*), indicando que a chamada ao método *d* no rastro 2 não tem contraparte no rastro 1. O último segmento apresenta a chamada ao método *e* alinhado nos dois rastros.

As Figuras 3 e 4 mostram fragmentos de rastros reais coletados da ferramenta ArgoUML representados em árvores, que será discutido na seção do estudo de caso. A Figura 3 apresenta parte da inicialização do sistema, especificamente a criação dos itens de *menu*. Observe que no fragmento apresentado todas as chamadas foram alinhadas. A Figura 4 mostra o desalinhamento a partir do momento que o usuário clica os botões "*create Deployment/Class*

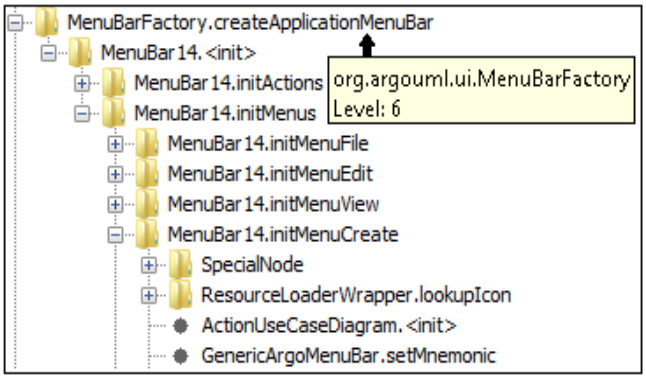

Figura 3. Parte da árvore que contém a inicialização.

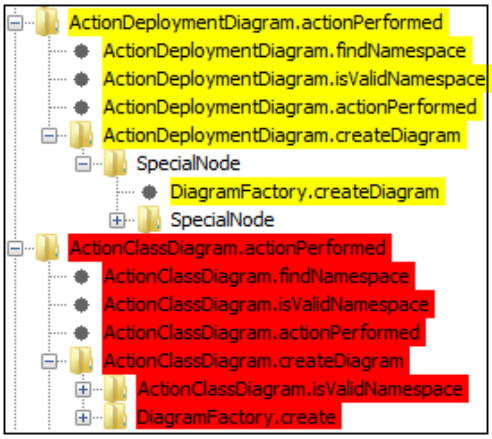

Figura 4. Parte da árvore que mostra a sequência de chamadas específicas para cada cenário.

diagram".

D. *Análise Visual do Alinhamento e Implementação da Lista de Adições/Alterações*

A análise para localizar métodos de uma subfuncionalidade específica começa após a construção da árvore de alinhamento. Com a análise visual sobre as árvores, alguns segmentos desalinhados podem ser descartados por não estarem semanticamente relacionados com a subfuncionalidade que está sendo avaliada, em especial os que estão no mesmo nível na árvore de um segmento considerado importante. O resultado final é uma árvore que contém somente os segmentos alinhados e os desalinhados que são importantes para a inserção ou alteração no código fonte.

Finalmente, uma lista de tarefas é elaborada com base nos segmentos selecionados, contendo sugestões de alterações no código, de inserções de novas classes, pacotes e/ou métodos.

III. AVALIAÇÃO DO TRABALHO

Este trabalho foi avaliado com um estudo de caso em um sistema real de grande porte. O sistema escolhido foi o ArgoUML 0.30 [15], uma ferramenta de modelagem UML, por se tratar um sistema relativamente complexo com aproximadamente 161.000 LOC, 92 pacotes, 2122 classes, 156 interfaces, 2314 atributos, 1914 atributos estáticos, 13918 métodos, 669 métodos estáticos. Nos números acima não foram somados o framework GEF - *Graph Editing Framework* e nem o metamodelo da OMG.

TABELA I. NÚMERO DE ELEMENTOS INSERIDOS PARA O DIAGRAMA DE OBJETOS

Tipo de Elemento	*Quantid.*
Pacotes	2
Classes	6
Métodos	13
Propriedades	3
Total de linhas de código inseridas	606

Esta seção apresenta os resultados obtidos na aplicação do método de alinhamento de sequências em rastros de execução sumarizados, com o objetivo de subsidiar a inclusão de novas funcionalidades no sistema.

Foram escolhidas duas funcionalidades a serem introduzidas no ArgoUML: o desenho de diagramas de objetos e o desenho de diagramas de pacotes. A seguir apresentaremos os resultados relativos ao primeiro em detalhes e os números relativos ao segundo.

A. *Inserção de Funcionalidades para Diagramas de Objetos*

A Tabela I mostra um resumo do número de elementos inseridos no código do sistema. As inserções somaram 606 linhas, nas quais estão incluídas linhas automaticamente inseridas pelo IDE Eclipse durante a implementação, tais como declaração de *imports*, métodos construtores sem parâmetro e sem corpo, métodos inseridos automaticamente após inserção da cláusula *implements*, dentre outros. A seguir serão apresentadas tabelas que mostram linhas mais específicas que foram inseridas no código, as quais somaram 269 linhas. A seguir apresentamos como os passos da abordagem foram aplicados.

1) Definição de Cenários de Execução: A escolha dos cenários similares foi baseada no critério de encontrar quais diagramas existentes se assemelhassem ao diagrama de objetos. Assim, os cenários escolhidos foram *"A. Desenhar Diagrama de Classes"* e *"B. Desenhar Diagrama de Componentes"*. O diagrama de componentes foi desenhado utilizando a paleta do diagrama de implantação (*deployment*). A Figura 2 mostra os cenários utilizados como referência para a criação do diagrama de objetos. Cada cenário para uma determinada funcionalidade foi definido incrementalmente com base no cenário anterior. Pode ser observado na Figura 2 que também existe uma correspondência direta entre as funcionalidades dos cenários A e B.

2) Sumarização dos Rastros: Para cada cenário foram coletados seis arquivos de rastros, um para cada *thread* lançada durante a execução. Primeiramente, foi aplicado o compressor nestes arquivos para eliminar laços e chamadas recursivas. Em seguida, foi executado o sumarizador nos rastros comprimidos, procurando retirar chamadas de métodos que não interferem na detecção dos métodos específicos de uma característica. As granularidades testadas para a sumarização foram 2, 3 e 4. Para a inserção do diagrama de objetos, as melhores árvores de alinhamento geradas foram com a granularidade 2, porque as outras com granularidade maior podaram alguns métodos específicos da funcionalidade em questão. A poda excessiva é indesejável porque não teria a cobertura suficiente para obtenção de informações direcionadas às tarefas de implementação da evolução. Isto implicaria na análise e depuração manual do

TABELA II. RESULTADOS DA COMPRESSÃO E SUMARIZAÇÃO DOS RASTROS DE EXECUÇÃO

	Número Funcion.	*Tamanho dos Rastros (chamadas de métodos)*	*Tamanho do Rastro Comprimido/ Taxa de Compressão / Tempo de Execução (ms)*	*Tamanho do Rastro Comprimido-Sumarizado / Taxa de Sumarização / Tempo de Execução (ms)*	*Tamanho do Rastro Comprimido-Sumarizado-Comprimido / Taxa de Compressão-Sumarização / Tempo de Execução (ms)*
Cenário A.3	5	243.589	169.782 / 30,29% / 18306	39.354 / 83,84% / 3388	35.356/ 85,48% / 2938
Cenário B.3	5	314.458	229.165 / 27,12% / 17808	52.592 / 83,27% / 4874	47.690 / 84,83% / 3749

sistema, o que é objetivo ser evitado para diminuir o esforço total. Logo, optou-se por uma cobertura maior em detrimento de uma possível baixa precisão. No passo final, o compressor é aplicado novamente sobre os rastros sumarizados para retirar possíveis laços não detectados na sua primeira execução. A Tabela II mostra o resultado de cada passo para reduzir os rastros dos cenários A.3 e B.3. A redução dos rastros apresentada está relacionada ao seu tamanho original.

3) Alinhamento dos Rastros: Nesta etapa foi aplicado o alinhamento nas *threads* sumarizadas obtidas no passo anterior, onde para cada alinhamento foi escolhido a *i*-ésima *thread* do cenário *A.j* e a *i*-ésima *thread* do cenário *B.j*. O resultado de cada alinhamento foi representado em uma árvore para a análise visual.

4) Análise Visual dos Rastros e Implementação: Por meio das árvores de alinhamento, algumas *threads* e alguns blocos desalinhados puderam ser descartados por não estarem semanticamente relacionados à funcionalidade analisada no momento. Por exemplo, no alinhamento da *thread* 1 do cenário A.1 e B.1 houve 25 desalinhamentos no nível 1 da árvore, após a análise restaram somente 4 blocos desalinhados que estão relacionados à criação dos diagramas de classe e de componentes. As *threads* 1 e 2 foram as selecionadas para auxiliar na implementação inicial do diagrama de objetos. Com a análise do alinhamento dos rastros referentes à *thread* 1 foi localizado no código fonte onde os itens de menu *New Class Diagram* e *New Deployment Diagram* são criados. Com a análise do alinhamento referente à *thread* 2 foram encontradas as chamadas de métodos que indicam a criação dos diagramas e as funcionalidades *desenhar uma classe* e *desenhar um componente*. Também foram encontradas as classes responsáveis por mostrar as figuras de uma *classe* e de um *componente* em um diagrama, bem como de uma *associação* e de uma *dependência*. Com a análise do alinhamento referente à *thread* 3 foram encontradas as chamadas de métodos das classes responsáveis por mostrar os gráficos de uma classe – `FigClass` - e de um componente – `FigMNode` - em um diagrama. Estas classes puderam ser descartadas pois já haviam sido consideradas na *thread* 2. Uma suposta classe `FigObject` que mostra os gráficos de um objeto já estava implementada no código e não foi necessário criá-la novamente, mas apenas utilizá-la. Com a

TABELA III. ALINHAMENTO ENTRE OS CENÁRIOS A.1 E B.1

Informações extraídas da *thread1*			
Localização	***Adição***	***Nº. de Linhas***	***Descrição da Obtenção da Informação para Implementação da Evolução***
`action.properties` (Arquivo)	`action.object-diagram` (Propriedade)	1	Pela árvore da Figura 3, foi possível localizar o método `initMenuCreate` responsável pela criação dos itens de menu. Para a inserção do item de menu *New Object Diagram*, foi necessário criar esta propriedade. Isto foi detectado através de observações no código fonte do respectivo método.
`menu.properties` (Arquivo)	`menu.item.object-diagram. mnemonic` (Propriedade)	1	O mesmo processo citado acima.
`ShortcutMgr` (Classe)	`ACTION_OBJECT_DIAGRAM` (Constante) e chamada de método usando esta constante e um objeto `ActionObjectDiagram` como parâmetro	2	O mesmo processo citado acima.
`org.argouml.ui` (Pacote)	Criação da classe `ActionObjectDiagram`	-	Um das folhas da Figura 3 mostra a criação de uma instância da classe `ActionUseCaseDiagram`. Continuando neste nível são apresentadas todas as outras instanciações para todos os diagramas existentes no sistema como `ActionClassDiagram` e `ActionDeploymentDiagram`.
`ActionObjectDiagram` (Classe)	Construtor usando o `action.object-diagram` acima criado como parâmetro	3	Este construtor foi criado pela observação no código fonte dos outros construtores das classes `ActionClassDiagram` e `ActionDeploymentDiagram` .
`MenuBar14` e `GenericArgoMenuBar` (Classes)	Inserção de comandos para a criação de um objeto `ActionObjectDiagram`, o `menu.item.object-diagram.mnemonic` e o `ACTION_OBJECT_DIAGRAM` no `JMenuItem` do diagrama de objetos, similares aos comandos já existentes para os outros diagramas.	5 para cada classe	Após a execução de todas as linhas acima, o item de menu foi criado no método `initMenuCreate` da classe `MenuBar14` – mostrado na Figura 3 - e foi alterado de maneira correspondente o método `initMenuCreate` da super classe `GenericArgoMenuBar` porque em `MenuBar14.initMenuCreate` é chamado o `super.initMenuCreate`.

análise do alinhamento referente à *thread* 4 foram encontradas as chamadas de métodos relacionadas às críticas e se optou por não implementar críticas para diagramas de objetos neste estudo de caso, pois eles poderiam ser implementados em uma segunda versão, sem interferir na avaliação do método proposto. Com a análise do alinhamento da *thread* 5, pudemos descartá-la porque o rastro foi todo alinhado, e isto significa que não possui chamadas de métodos de funcionalidades específicas do cenário. Com análise na *thread* 6 os segmentos só apresentam métodos relacionados a OCL e críticas, e foram descartadas por motivo semelhante à *thread* 4.

Após a análise do significado das *threads*, o processo de implementação da funcionalidade consistiu de quatro etapas:

a) Etapa 1: Por meio da árvore de alinhamento da *thread* 1 do cenário A.1 com a *thread1* do cenário B.1 foi localizada a inicialização do sistema. A hipótese é que os segmentos alinhados na árvore mostram os elementos de inicialização os quais são comuns para todos diagramas e portanto tem que ser adaptados para inicializar elementos relativos ao novo diagrama a ser criado. Para inserir a opção de criar o diagrama de objetos no item de menu "*Create"* e na barra de ferramentas, o modo de navegação na árvore de alinhamento para a obtenção das informações necessárias a esta implementação foi uma expansão seletiva dos nós da árvore buscando elementos semanticamente relevantes em nós mais profundos. O critério para definir o que é um elemento semanticamente relevante foi baseado nos próprios nomes das classes e métodos. Com a expansão do primeiro nó da árvore, já foi possível encontrar a chamada `initializeSubsystems` da classe `Main` e com a expansão deste nó – nível 2 – foi exibido o método `initializeGUI` da classe `Main` – nível 3. O processo de navegação foi o mesmo até chegar ao nível 6, onde foi encontrado o método `createApplicationMenuBar` da classe `MenuBarFactory` – nó raiz da Figura 3. Este método faz chamada ao construtor da classe `MenuBar14` – nível 7, apresentado na última linha da Tabela III, e onde devem ser realizadas as alterações para incluir o menu *"New Object Diagram"*. A Figura 3 também mostra um trecho da árvore, responsável pela criação da barra de menu e seus respectivos itens, onde é possível ver o pacote em que a classe está e qual método é responsável pela criação do item de menu *"Create"* - `MenuBar14.initMenuCreate`. Com a análise de um nível abaixo pode-se ver a inicialização da classe `ActionUseCaseDiagram.<init>`. O nó `SpecialNode` (não expandido na figura por razões de espaço) contém chamadas da classe `GenericArgoMenuBar` que é uma super classe de `MenuBar14` – a informação de super classe foi obtida pela leitura no código.

A Tabela III mostra a ordem de execução das tarefas necessárias para incluir a opção *New Object Diagram*. Na tabela são mostrados os pontos de localização das alterações (adições), o elemento adicionados, um dimensionamento deste elemento em termos de número de linhas, e uma descrição que justifica como a árvore de alinhamento forneceu a informação precisa para dar suporte à modificação. As demais tabelas relativas às próximas etapas têm estrutura similar.

b) Etapa 2: *No alinha*mento da *thread2* do cenário A.1 com a *thread2* do cenário B.1 foram encontradas chamadas de métodos fundamentais para a funcionalidade criar diagrama de objetos. A análise da árvore alinhada para esta etapa foi feita de maneira diferente da etapa anterior. Neste caso para detectar quais são as chamadas de métodos específicas de cada cenário, é necessário analisar os segmentos de desalinhamento na árvore. A hipótese é que os segmentos de desalinhamento contêm trechos específicos para cada diagrama e que, portanto deveriam ser inseridos

TABELA IV. ALINHAMENTO ENTRE OS CENÁRIOS A.1 E B.1

Informações extraídas da *thread2*			
Localização	***Adição***	***Nº. de Linhas***	***Descrição da Obtenção da Informação para Implementação da Evolução***
`ActionObjectDiagram` (Classe)	Métodos: `createDiagram` e `isValidNamespace`	21	Com análise na árvore de alinhamento foi possível criar estes métodos. O método `findNamespace` utilizado foi da super classe. O fragmento da árvore na Figura 4 mostra estas chamadas.
`label.properties` (Arquivo)	`label.object-diagram`	1	Esta propriedade foi criada para ser utilizada pelo método `getLabelName` citado na 6ª linha. Esta propriedade foi criada através de observações no código fonte.
`org.argouml.images` (Pacote)	ObjectDiagram.png	-	Ícone criado para inserir imagem no item de menu e na barra de ferramentas. A chamada deste ícone é feita pelo construtor da classe `PropPanelUMLObjectDiagram` que será comentada nesta tabela.
`app` (Módulo)	Pacotes: `org.argouml.uml.diagram.object` e `org.argouml.uml.diagram.object.ui`	-	Construção dos pacotes seguindo o padrão do sistema para a inserção de classes específicas do diagrama de objetos. A informação foi obtida analisando os desalinhamentos na respectiva árvore.
`org.argouml.uml.diagram.object.ui`	Classes: `UMLObjectDiagram`, `PropPanelUMLObjectDiagram` e `InitObjectDiagram`	-	Classes criadas através dos desalinhamentos verificados na análise da árvore e inseridos no pacote citado acima.
`UMLObjectDiagram`	O método `getLabelName` foi implementado e o `setNamespace` adicionado	16	A classe `UMLObjectDiagram` herda os métodos da classe abstrata `UMLDiagram` e o `getLabelName` é um destes métodos. Foi criado pela análise da árvore de alinhamento.

TABELA V. ALINHAMENTO ENTRE OS CENÁRIOS A.2 E B.2

Informações extraídas da *thread2*				
Localização (Pacote)	***Nova Classe***	***Inserção de métodos***	***Nº. de Linhas***	***Descrição da Obtenção da Informação para Implementação da Evolução***
`org.argouml.uml.diagram.object.ui`	`ObjectDiagramRenderer`	`getFigNodeFor`	10	Classe e método criados por meio de análise da árvore de alinhamento. Os métodos de funções semelhantes foram encontrados no primeiro nível da árvore.
`org.argouml.uml.diagram.object`	`ObjectDiagramGraphModel`	`candAddNode` e `addNode`	29	Esta situação é a mesma do caso citado acima. A classe `ObjectDiagramGraphModel` define uma ponte entre a representação do meta-modelo UML do projeto e a interface `GraphModel` usada pelo GEF.
`org.argouml.uml.diagram.object.ui`	`ObjectDiagramPropPanelFactory`	`createPropPanel`	5	Classe e método criados por meio de análise da árvore de alinhamento. Este método utiliza PropPanelUMLObjectDiagram criado na Tabela IV – análise realizada no código fonte.

respectivos trechos similares para o novo diagrama de objetos. A Figura 4 mostra uma parte da árvore destacando as chamadas específicas de cada cenário. Os métodos de cada cenário são apresentados na árvore com cores diferentes, sendo fácil a localização dos mesmos. A Tabela IV mostra o que foi implementado no sistema. O processo de análise das árvores de alinhamento das próximas etapas é equivalente ao desta etapa.

c) Etapa 3*:* Nos cenários A.2 e B.2 foram introduzidas as funcionalidades "3. Inserir classe C1" e "3. Inserir componente D1", respectivamente. Neste caso, a hipótese é que desalinhamentos da árvore mostram o que é específico para o conceito "classe" e o que é específico para o conceito "componente", logo indica o que deverá ser adaptado para o conceito "objeto". Após alinhamento da thread2 do cenário A.2 com a thread2 do cenário B.2 foram encontradas as chamadas de métodos usadas para inserir uma classe e um componente nos seus respectivos diagramas. A Tabela V mostra as tarefas em ordem de execução para permitir a inserção de um objeto no diagrama de objetos.

d) Etapa 4: Nos cenários A.3 e B.3 foram introduzidas funcionalidades a respeito do conceito de *"associações /dependências entre classes/componentes"*. Assim como na etapa 3, os desalinhamentos devem mostrar o que deve ser adaptado para implementação de "*associações entre objetos*". No alinhamento da *thread2* do cenário A.3 com a *thread2* do cenário B.3 foram localizadas as chamadas de métodos usadas para desenhar a composição e a dependência entre seus respectivos elementos dos diagramas desenhados na execução dos cenários. A Tabela VI mostra as tarefas na ordem de execução para inserir a funcionalidade instância de uma associação entre dois objetos.

B. Análise de Alinhamento Pós-Implementação:

Após a nova funcionalidade *"Desenhar diagrama de objetos"* ter sido concluída, foi realizada uma análise para verificar a semelhança dos rastros deste novo cenário. Foi efetuado um alinhamento com os rastros do cenário B3 para

TABELA VI. ALINHAMENTO ENTRE OS CENÁRIOS A.3 E B.3

Informações extraídas da *thread2*			
Localização	***Adição***	***Nº. de Linhas***	***Descrição da Obtenção da Informação para Implementação da Evolução***
`org.argouml.uml.diagram.deployment.ui` (Pacote)	As classes `FigObject` e `SelectionObject` foram movidas para `org.argouml.uml.diagram`	-	Através da árvore de alinhamento seria necessário criar estas classes. No entanto, as mesmas já estavam criadas, específicas para o diagrama de implantação.
`DiagramFactory` (Classe)	Inserção de uma condição para o diagrama de objetos no método `createDiagram` e uma inserção na tabela *hash* no construtor	5	Após a localização deste método na árvore, estas linhas de códigos foram inseridas por meio de análise do código.
`ObjectDiagramGraphModel` (Classe)	Métodos adicionados: `canAddEdge` e `addEdge`	51	Classe e método criados através de análise da árvore de alinhamento. Os métodos de funções semelhantes foram encontrados no 3º nível da árvore. Estudo do metamodelo para descobrir que um *edge* é um objeto `Link` do metamodelo OMG da UML.
`ObjectDiagramRenderer` (Classe)	`getFigEdgeFor`	50	Classe e método criados através de análise da árvore de alinhamento. Os métodos de funções semelhantes foram encontrados no 4º nível da árvore.
`core-model` (Módulo)	Criado a interface `ObjectDiagram`	2	Criado por análise no código fonte.
`UMLObjectDiagram` (Classe)	Métodos implementados: `relocate`, `isRelocationAllowed`, `getRelocationCandidates`, `createDiagramElement`, `getUmlActions`, `doesAccept`, `encloserChanged`	43	Os métodos `createDiagramElement` e `doesAccept` foram encontrados no nível 2 e 3 da árvore. A implementação dos demais é devido a herança da classe abstrata UMLDiagram.
`UMLObjectDiagram` (Classe)	Métodos adicionados: `getActionLink`, `getActionObject`, `createGraphModel`	19	O método `createGraphModel` foi criado para ser utilizado no construtor da classe `UMLObjectDiagram`.

"Desenhar diagrama de implantação". Na *thread* 4, houveram poucos desalinhamentos e estes foram somente nos rastros do diagrama de implantação, indicando que neste há mais funcionalidades implementadas, o que não afeta o objetivo do estudo que foi avaliar a redução de esforço na implementação da funcionalidade básica. Estes desalinhamentos indicam inclusive novas possíveis evoluções que podem ser efetivadas na funcionalidade recém-incluída.

IV. DISCUSSÃO

Após a implementação das evoluções desejadas, o software foi testado e considerado bastante adequado funcionalmente. Isto demonstra que as alterações propostas tiveram um grau de correção bastante satisfatório.

A questão mais importante a ser discutida neste trabalho é se houve ou não redução de esforço ao se inserir uma nova funcionalidade em um sistema de código disponível. A rigor, uma resposta definitiva para esta pergunta iria requerer um estudo experimental mais aprofundado onde o esforço seria medido e avaliado, considerando uma amostra significativa de desenvolvedores. Logo, esta é uma limitação deste trabalho, pois não é possível responder qual o tamanho da redução do esforço necessário no estudo específico.

Entretanto, o estudo mostrou claramente por meio das tabelas apresentadas que as modificações/adições introduzidas na implementação foram obtidas de uma maneira direta, ou com um forte direcionamento para um estudo de pequenos trechos do código fonte, a partir primordialmente das informações contidas na árvore de alinhamento. Uma exceção a este ponto foi que no estudo foi necessário recorrer à análise do metamodelo OMG da UML para descobrir como representar uma associação de objetos, sendo identificado que a classe a ser utilizada chama-se `Link`. Cabe ressaltar que de qualquer forma, o alinhamento forneceu uma diretriz sobre o que procurar na documentação do metamodelo. Além disso, o trabalho específico de análise da árvore de alinhamento e definição dos pontos a serem alterados na implementação foi realizado somente pela primeira autora e durou apenas cerca de um dia. Relembrando que o código fonte do ArgoUML contém 92 pacotes, 2122 classe, 13918 métodos, é razoável considerar, que o trabalho de entendimento e localização dos pontos de modificação mostrados nas tabelas anteriores utilizando somente a leitura da documentação existente, inspeção e depuração do código duraria muito mais que isto. Logo, se considerarmos que a experiência da autora no projeto e implementação do ArgoUML era praticamente inexistente, podemos considerar a redução de esforço extremamente expressiva. Logo, sob este aspecto podemos considerar que a abordagem foi bastante bem sucedida ao diminuir drasticamente o esforço para alteração.

Um possível questionamento sob o estudo realizado é que a semelhança entre *Desenhar Diagrama de Classes, Componentes, Objetos e Pacotes* é grande e por isto de certa forma o trabalho de evolução se tornou mais simplificado. De certa forma, isto é verdade. Por outro lado, mesmo assim a tarefa de evolução é considerada relevante e que poderia ocorrer na prática em ambientes de desenvolvimento. Além do mais, mesmo que as tarefas não fossem completamente semelhantes, normalmente existem muitos pontos em comum para qualquer funcionalidade em sistemas interativos, tais como, a localização de trechos de código para alteração de menus, acesso ao painel central, interligação entre os vários componentes da aplicação. Na realidade, uma das grandes dificuldades de se efetivar atualizações em um sistema de grande porte é que os elementos da aplicação são fortemente relacionados e descobrir estas relações no código fonte a partir da observação externa não é trivial, e a abordagem se mostrou efetiva neste sentido, basta identificar nas tabelas apresentadas anteriormente a variedade de classes e pacotes envolvidos na evolução.

Um ponto sobre o uso de rastros de execução para compreensão de programas, e em especial, alinhamento de rastros de execução de aplicações interativas é que os rastros obtidos na execução de cenários idênticos podem não ser exatamente idênticos devido a pequenas variações na interação do usuário durante a coleta do rastro. Nestes casos, podem ocorrer desalinhamentos nos rastros de execução que não estão relacionados com a nossa hipótese de que os desalinhamentos são devidos a diferenças nas funcionalidades exercitadas. Logo, pode haver ruídos nos rastros de execução que deverão ser descartados pelos desenvolvedores. De fato, em um trabalho prévio [11] de alinhamento de rastros de execução para compreensão de sistemas, o número de segmentos a serem analisados e descartados é muito grande, tornando o esforço de compreensão igualmente alto. O uso da técnica de sumarização proposta neste trabalho permitiu uma drástica redução no tamanho dos rastros (de ~30% para ~80%), conforme apresentado na Tabela 2. Esta diminuição no tamanho dos rastros associada a um adequado formato de árvore para a apresentação do alinhamento, permitiu com que a abordagem estivesse menos sensível a ruídos e, portanto atingindo mais plenamente seu objetivo.

Outro aspecto que pode ser ressaltado como resultado do estudo é que durante a introdução das modificações foram encontradas porções de código redundantes na implementação dos *menus*. Por exemplo, o código original do sistema que faz a inserção de um item de *menu* é sujeito a severas críticas. O itens de *menu* são introduzidos na superclasse `GenericArgoMenuBar` e logo em seguida retirados e inseridos novamente pela subclasse `MenuBar14`. Além de ser um projeto de baixa qualidade tem um impacto na eficiência do código. Logo, um trabalho futuro pode estudar a possibilidade de aplicar a técnica de alinhamento de rastros de execução para identificar trechos de código de baixa qualidade.

V. TRABALHOS RELACIONADOS

Pelo o melhor do nosso conhecimento, não existe aplica-

ção de algoritmo de alinhamento de sequências para análise de rastros de execução, além daqueles desenvolvidos pelos autores [8], o qual mostrou e introduziu a hipótese de uso de alinhamento de rastros de execução, mas não a comprovou com um estudo detalhado de evolução de software.

Outras técnicas para análise de rastros de execução têm sido extensivamente utilizadas em compreensão de programas [1, 16]. Alguns autores [9, 13] sugerem o uso integrado de visões estáticas e dinâmicas do sistema de software. As visões dinâmicas são obtidas por meio de *profiling* das funcionalidades mais usadas no sistema. Entretanto, o objetivo primário é obter a arquitetura do sistema para facilitar o entendimento do mesmo. Em nosso caso, focalizamos não em funcionalidades mais usadas para compreender o sistema de uma forma geral, mas sim em funcionalidades bem escolhidas para prover a informação diretamente customizada para processos de evolução de software, o que requer menor esforço para efetivar as alterações de código necessárias.

As ferramentas de depuração e busca dos IDEs ajudam a entender como que objetos interagem uns com os outros. Entretanto, não se consegue extrair uma visão abrangente de interdependência entre diversas classes envolvidas na implementação de uma funcionalidade como nossa abordagem demonstrou extrair. Assim, a abordagem de evolução apresentada aqui direciona o trabalho do desenvolvedor de uma maneira bastante precisa, e conseqüente isto reduziu drasticamente o esforço.

Um trabalho que de certa forma se assemelha ao nosso, baseia-se na premissa de que a inserção de novas funcionalidades semelhantes em um sistema é vista como a prática de fazer cópias de artefatos, ou seja, clones [6]. Outros estudos mostram que de 7% a 23% de código em sistemas típicos são clonados[12]. De certa maneira, isto corrobora com nossa hipótese de que é útil evoluir código fonte utilizando trechos de código semelhantes.

VI. CONCLUSÕES

Este trabalho apresentou uma abordagem baseada na análise de rastros de execução para auxiliar a evolução de software. Uma das hipóteses que trabalhamos foi que os desalinhamentos de rastros de execução são pontos de implementação específicos a uma determinada funcionalidade que se deseja compreender. A hipótese ainda estabelece que quando se deseja introduzir uma funcionalidade em um sistema que seja semelhante a uma funcionalidade pré-existente, então os pontos de desalinhamento da funcionalidade original dão fortes indícios de onde deve ser introduzida a nova funcionalidade. Estas hipóteses foram confirmadas no estudo de casos que introduziu dois novos diagramas na ferramenta de modelagem ArgoUML. Outra hipótese é que o esforço para implementação das evoluções seria menor com a abordagem proposta. Apesar de não termos conduzido um estudo experimental robusto para comprovar esta hipótese, o tempo de elaboração da evolução (estudo e implementação), que foi de aproximadamente um dia de trabalho da primeira autora, foi considerado extremamente relevante, especialmente se comparado com abordagens tradicionais e em uma situação onde o desenvolvedor não tem conhecimento prévio a respeito da implementação do sistema, em especial um sistema de mais de 160.000 linhas de código.

Dentre os trabalhos futuros, podemos citar um estudo da aplicação da técnica de alinhamento de rastros sumarizados na detecção e reprojeto de trechos de código de baixa qualidade.

AGRADECIMENTOS

Agradecemos à Capes e ao CNPq pelo apoio parcial a esta pesquisa. Agradecemos aos diversos membros do grupo de pesquisa que tem contribuído nos projetos de Engenharia Reversa de Software do Programa de Pós-Graduação em Ciência da Computação da UFU.

REFERÊNCIAS

[1] B. Cornelissen, A. Zaidman, A. van Deursen, A Controlled Experiment for Program Comprehension through Trace Visualization, *IEEE Trans. on Software Engineering*, vol. 99, no. PrePrints, 2010

[2] A. de Lucia, R. Oliveto, G. Tortora, Assessing IR-based traceability recovery tools through controlled experiments, *Empirical Software Engineering*, 14 (1), Springer, Netherlands, 2009, pp. 57-92.

[3] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[4] A. Hamou-lhadj and T.C. Lethbridge. Compression Techniques to Simplify the Analysis of Large Execution Traces. In *Proc. of IWPC*, 2002, pages 159-168.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold. An Overview of Aspect J. In *Proc. of ECOOP*, 2001.

[6] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. *In IEEE Intl. Symp. on Empirical Software Engineering*, pp. 83–92, 2004.

[7] T. C. Lethbridge, J. Singer, and A. Forward, How Software Engineers use Documentation: The State of the Practice, *IEEE Software,* vol. 20, no. 6. CA, USA: IEEE Computer Society Press, 2003, pp. 35–39.

[8] M. Maia, V. Sobreira, K. Paixão, S. de Amo, I. Silva. Using a Sequence Alignment Algorithm to Identify Specific and Common Code from Execution Traces. *4th Intl. Workshop on Program Comprehension through Dynamic Analysis*. Antwerp, pp. 6-11. 2008.

[9] M. Mit and M. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *Proc. of ICSE*, 2003, pp 24-27.

[10] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology,* (48):443–453, 1970.

[11] Paixão, K.R. *Alinhamento de rastros de execução de programas para compreensão de pontos de variação em código-fonte*. Dissertação de Mestrado. Universidade Federal de Uberlândia. 2009. 78pp.

[12] C.K. Roy and J.R. Cordy, An Empirical Study of Function Clones in Open Source Software Systems, in: *Proc. of the 15th Working Conference on Reverse Engineering, WCRE 2008*, pp. 81-90 (2008).

[13] K. Sartipi, and N. Dezhkam. An Almalgamated Dynamic and Static Architecture Reconstruction Framework to Control Component Interactions. In *Proc. of WCRE*, 2007, pp 259-268.

[14] V. Sobreira, and M. Maia. A Visual Trace Analysis Tool for Understanding Feature Scattering. In *Proc. of WCRE*, 2008, pp.337-338.

[15] UML modeling tool: http://argouml.tigris.org/ last accessed in 08/23/2010

[16] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. *In Proc. of ICSE,* 2004, pp. 470-479.