

UM ALGORITMO PARA CÁLCULO DE DISTÂNCIA ENTRE DIAGRAMAS DE SEQUÊNCIA DA UML

Joni Scaravonatti Canal*

Resumo: O cálculo de distância entre diagramas UML é de extrema importância para correta composição de modelos, deixando clara a quantidade mínima de edições que um modelo terá para ser composto por outro. Porém, a busca dessa similaridade por meio do cálculo de distância ainda é um grande desafio para os desenvolvedores, visto que as ferramentas hoje disponíveis não apresentam algoritmos capazes de efetuar a análise dessa diferenciação entre os elementos dos diagramas. Este trabalho, portanto, foca em propor um algoritmo de cálculo de distância entre diagramas de sequência para flexibilizar a identificação de similaridade entre modelos. Para isso, o algoritmo proposto foi implementado na ferramenta de composição MoCoTo, utilizada como um plug-in da plataforma eclipse, através da utilização de padrões de projeto e do algoritmo de Levenshtein Distance. Os resultados obtidos mostram a efetividade do algoritmo através de cenários de evolução e, também, a identificação da possível necessidade de um estudo aprofundado quanto à utilização de um padrão de tratamento que poderá ser feito com o resultado desse algoritmo, para auxiliar os desenvolvedores no andamento de uma composição de modelos.

Palavras-chave: UML. Distância entre Modelos. Algoritmo Levenshtein. Composição de Modelos.

* Aluno concluinte do curso de Análise de Desenvolvimento de Sistemas. E-mail: jonicanal@hotmail.com

1 INTRODUÇÃO

A busca pela distância (ou similaridade) dentre dois modelos tem um papel importante dentro da composição de modelos de software. A correta identificação da similaridade evita que um modelo gerado de uma composição tenha problemas de incompatibilidade ou inconsistências, visto que através desse resultado a ferramenta utilizada poderá tratar o nível de edição necessário para compor os elementos. Consequentemente ter total controle das operações necessárias para tal desenvolvimento.

O cálculo da distância entre dois modelos de sequência é de fundamental importância para a atividade de composição de modelos, a qual pode ser definida como sendo um conjunto de atividades a serem executadas sobre dois modelos de entrada, M_A e M_B , com o objetivo de produzir um modelo desejado, M_{AB} (FARIAS et. Al., 2013). Porém, os conflitos entre os elementos dos modelos que formam o M_A e o M_B , acabam sendo resolvido de uma forma inadequada, principalmente pelo não conhecimento do nível de similaridade dos dois modelos para a correta identificação do que deve ser realizado e implementado. Consequentemente, um modelo composto com inconsistência é produzido, M_{CM} (FARIAS et. Al., 2014).

Com a necessidade de resolver esses conflitos da melhor forma, muitas ferramentas são apresentadas com o objetivo de tornar rápida e clara a identificação da similaridade dos modelos, auxiliando os desenvolvedores a terem total conhecimento das operações necessárias dentre os dois modelos para uma composição. Para tal intuito, podem-se citar duas ferramentas: a *Umbrello UML* (UMBRELLO, 2014) uma ferramenta *open-source* para utilização em LINUX; e a ferramenta *Astah* (ASTAH, 2014).

Embora várias ferramentas e técnicas tenham sido propostas (ECLIPSE, 2012), bem como estudos sobre composição de modelos tenham sido executados (FARIAS et al., 2012) (FARIAS et al., 2010), esse cálculo de distância entre dois diagramas de sequência da UML ainda é um problema em aberto.

O principal motivo é a ausência de algoritmos capazes de detectar, por exemplo, as semelhanças entre as mensagens trocadas entre os objetos e a ordem de execução de tais mensagens. Além disso, não é possível quantificar o quão similares (ou diferentes) os elementos do diagrama são. De fato, as ferramentas atuais pecam ao fornecer esta funcionalidade, ou por não serem capazes de

identificar as similaridades, ou por não conseguirem identificar as diferenças e os conflitos.

A falta de capacidade em identificar a similaridade parte também do conceito da mesma, que acaba sendo complexo tendo em vistas as diversas “estruturas” que devem ser estudadas. Logo, para que essa busca tenha sucesso na sua realização, cada parte da divisão citada deve ser avaliada, criando uma espécie de passo-a-passo para que, no seu final, chegue à um denominador comum.

Este trabalho, portanto, propõe um algoritmo que trata o primeiro passado dessa estrutura de similaridade, que é o cálculo de distância entre diagramas de sequência UML, com o objetivo de mitigar os problemas identificados anteriormente. O algoritmo foi implementado na ferramenta MoCoTo, utilizada como um plug-in da plataforma eclipse, através da utilização de padrões de projeto e do algoritmo de *Levenshtein distance* (LEVENSHTEIN,2014). Os resultados obtidos indicam que o algoritmo proposto foi efetivo para o cálculo da distância entre diagramas de sequência que passaram por cenários de evolução.

O trabalho é organizado da seguinte forma. A seção 2 apresenta a fundamentação teórica, a qual descreve os conceitos necessários para o entendimento da técnica cálculo de similaridade, bem como da utilização do algoritmo com a ferramenta. A Seção 3 apresenta o projeto da ferramenta proposta e as principais decisões tomadas para permitir a implementação do algoritmo. A Seção 4 apresenta o resultado da avaliação do cálculo de similaridade proposta e do algoritmo implementado. A Seção 5 apresenta os trabalhos relacionados. E, por fim, a Seção 6 apresenta as considerações finais e os trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo foca na descrição dos conceitos que entrarão como base no desenvolvimento da ferramenta. Para tal, a Seção 2.1 apresenta o conceito da composição de modelos. Seção 2.2 o conceito de distância entre modelos. Por fim, a Seção 2.3 apresenta o conceito do diagrama de sequência.

2.1 Composição de Modelos

As empresas de tecnologia focadas em sistemas têm, como principal dificuldade, a organização quanto ao desenvolvimento de softwares de grande porte. Para isso, uma equipe de tamanho considerável é contratada com um mesmo foco principal, subdivididas em focos secundários que são de responsabilidade de cada equipe e que devem estar mesclados no software para um correto desenvolvimento do foco principal. Essa divisão acaba limitando o conhecimento da equipe para seus focos secundários, interferindo na correta interpretação de conflitos em uma composição de modelos.

Essa composição deve mesclar corretamente dois modelos diferentes que afetem a mesma lógica de desenvolvimento. A composição modelos tem como objetivos evitar interpretações erradas e definir o papel de cada modelo no mecanismo de composição (OLIVEIRA, 2008). As ferramentas de composição tem nesse quesito seu principal foco de correção, fazendo com que as inconsistências em sua total existência sejam identificadas e, que sejam passados os corretos passos a seguir para correção e término dos conflitos, evitando problemas na lógica de programação, na estrutura do código e, por fim, gerando o correto modelo mesclado.

Uma composição de modelo pode ter como entrada dois modelos diferentes M_A e M_B . Através do uso de uma ferramenta, e de estratégias bem definidas, a composição de tais modelos deve gerar um modelo composto M_{AB} . Porém, caso tal estratégia não esteja devidamente definida ou a ferramenta de composição em si utilizada não tenha a necessária função, o modelo composto a ser gerado é um totalmente diferente do pretendido inicialmente e com diversas inconsistências, seguindo o mesmo exemplo, um modelo M_{CM} .

De tal forma, para que essa composição seja bem executada, o cálculo de distância (similaridade) entre os diagramas à serem comparados deve estar bem definida. Um dos problemas encontrados nas ferramentas de composição é na identificação dessa distância, visto que elas não estão preparadas para tratar essas comparações, principalmente quando se diz respeito aos elementos que fazem parte de um diagrama. Esse valor acaba se perdendo, causando retrabalhos aos desenvolvedores, pelo fato de terem em si a preocupação maior da composição como um todo, ao gerar o modelo M_{AB} , e não a preocupação de tratar essa composição de acordo com o nível de dificuldade para com as edições necessárias

ao comparar um modelo M_A com um M_B , nos quais podem ser inserção, deleção e adição.

2.1.1 Ferramentas de Composição

Ferramentas de composição são caracterizadas de acordo com sua forma de desenvolvimento e operação. Essa caracterização é identificada através de técnicas desenvolvidas de acordo com a necessidade de desenvolvimento de um software. As ferramentas mais utilizadas atualmente são as que referenciam a técnica baseada em heurística como a IBM RSA (IBM, 2008), e as que focam nas técnicas baseadas em especificação, no caso da *Epsilon* (ECLIPSE, 2012).

2.1.1.1 IBM RSA

A IBM RSA é uma ferramenta de design, modelagem e desenvolvimento através do UML, focado em projetos de linguagem WEB e JAVA. Construído com base no software de desenvolvimento Eclipse, essa ferramenta tem uma variedades de plug-ins de tal utilização, além de extensões que atuam diretamente no aprimoramento da ferramenta.

Com um completo suporte e disponibilidade de ferramentas para modelagem UML, além de possibilidades de estruturação não especificamente física, ou seja, com uma disponibilidade de serviços em nuvem, a utilização da IBM acaba impactando no retorno de seu investimento, por se tratar de uma plataforma confiável e flexível. Apesar disso, sua utilização não passa por um processo automático, ou seja, dando a responsabilidade de interpretação de conflitos ao usuário, ficando frágil para com suas correções.

2.1.1.2 Epsilon

A ferramenta em Epsilon trata-se de uma família de linguagens e ferramentas para a geração de código, transformação de modelo-a-modelo, validação do modelo e comparação. Com o desenvolvimento baseado no EMF, essa ferramenta possibilidade à integração com sistemas Eclipse, através também de sua grande variedade de *plug-ins* para tal plataforma.

A possibilidade de reutilização do código (linguagem de expressão comum) e a alta disponibilidade de documentação e plug-ins fazem com que a Epsilon ofereça a possibilidade de uma extensão da ferramenta, através de um suporte e de desenvolvimento próprios de acordo com a necessidade da empresa e do software em questão.

2.2 Distância entre Modelos

Para que uma composição de modelos gere um novo modelo com o mínimo possível de inconsistências, é necessário que uma estratégia de tal operação seja criada antes de interligar dois modelos. A mesma pode ser gerada através do resultado ocorrente no cálculo de distância entre os dois modelos, que nada mais é que a identificação do quanto será necessário modificar (em nível de quantidade de operações) de um modelo para outro no momento dessa composição.

2.2.1 Levenshtein distance

A Levenshtein distance (LEVENSHTEIN, 2014) é uma função baseada em caractere e serve com base para o cálculo da função de similaridade de mesmo nome, e é dada pelo menor número de operações necessárias para transformar uma String em outra. (SILVA, 2007)

A utilização desse método passa por um cálculo que se baseia em um estudo de comparação de Strings, no qual se utiliza uma matriz $(n + 1) \times (m + 1)$, onde n e m são os números de caracteres dos dois Strings. Ou seja, esse algoritmo compara dois Strings através do cálculo de operações necessárias para a modificação de uma para outra, tratando assim suas inserções, deleções e edições necessárias.

Por exemplo, pegando duas Strings, CLASSE e CASO, e aplicando o algoritmo de Levenshtein distance, pode-se identificar qual o nº mínimo de operações necessária para que a primeira (classe) se transforme na outra (caso), conforme se é observado na Figura 1.

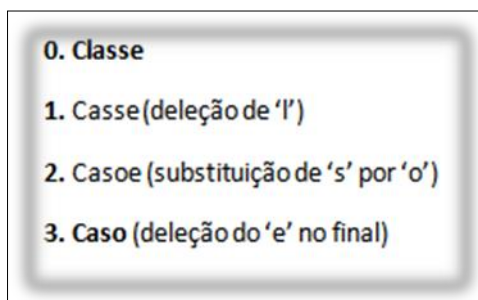


Figura 1. Exemplo de Cálculo de Distância entre Strings

Fonte: Próprio Autor

A partir do exemplo dado, pode-se identificar que a distância entre as duas Strings comparadas, através do cálculo de Levenshtein, é de 3. Isso porque ocorre primeiramente a deleção de um caractere (caso 1), depois a substituição de outro (caso 2) e, por fim, a deleção de outro caractere (caso 3). Esse algoritmo é preparado para utilizar o menor caminho possível em comparação à outros possíveis caminhos para gerar a String resultante.

Através disso, em uma comparação de diagramas onde não se tem uma estrutura inteira de Strings, é feito um tratamento de varredura entre seus elementos para transformá-los no formato necessário para geração do cálculo. Ou seja, utilizando um diagrama de sequência como exemplo, se verifica todos seus elementos (*lifelines()*, mensagens) identificando qual-a-qual, e os transformando em Strings. Dessa forma, se é utilizado o algoritmo para fazer o cálculo de distância entre esses Strings e conseqüentemente dos modelos em si comparados.

Como esse cálculo gerado, o desenvolvedor poderá ter total conhecimento da dificuldade que irá encontrar na composição dos diagramas e assim escolher a correta estratégia para tratar tal situação e evitar que um novo diagrama seja gerado com inconsistências, interferindo em um desenvolvimento futuro e trazendo um retrabalho em nível de composição.

2.3 Diagrama de Sequência

Um diagrama de sequência representa as mensagens passadas entre os objetos de um programa em desenvolvimento, em sequência de tempo de iteração. Essa distribuição pelo tempo faz com que o diagrama de sequência crie um cenário

individual de cada objetivo e participante das iterações, identificando cada papel dos mesmos, por meio de setas entre as linhas de vida.

Por meio de operações ou métodos, essas iterações identificadas no diagrama de sequência vieram a partir de um diagrama de casos de uso, definindo inicialmente o papel do sistema como um todo para depois definir as operações e papéis de cada responsável no programa. Ou seja, o diagrama de sequência serve para identificar cada mudança ou implementação realizada em um caso de uso, por meio de mensagens e iteração que tal modificação terá em seu papel.

O diagrama de sequência funciona através de uma forma bidimensional, onde as informações em vertical no gráfico do diagrama fazem referencia ao eixo de tempo, e as horizontais, ou em certo momento diagonais, referenciam os objetos que fazem a troca de informações e representam as mensagens trocadas entre objetos, tendo a correta identificação de seus nomes, parâmetros e condições para tal iteração. Existem também as mensagens de retorno, identificadas por linhas horizontais tracejadas.

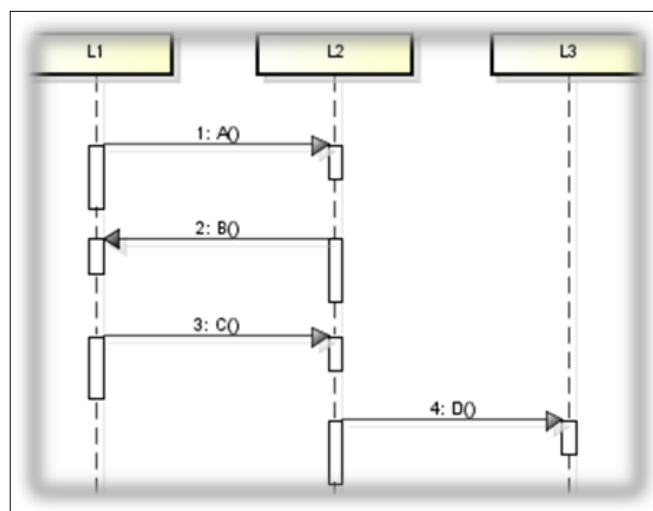


Figura 2. Exemplo de Diagrama de Diagrama de Sequência

Fonte: Elaborado pelo Autor

A figura 2 apresenta um exemplo de diagrama de sequência. No caso, podem-se identificar as lifelines (L1, L2 e L3) e as mensagens trocadas entre elas (A, B, C e D). Para tal, todas as Features deverão estar definidas, tratando-se então das sequências de atividades que ocorrerão no MoCoTo-SPL desde a inicialização do

software até seu término, através da realização dos casos de uso. O tratamento de um diagrama de sequência na ferramenta MoCoTo-SPL passa pelas operações de análise, comparação (focada neste trabalho), composição, avaliação e persistência.

Além disso, se é utilizado um metamodelo UML, usado para especificar o modelo que compreende o UML, com o papel de definir a semântica para modelar elementos dentro de um modelo que está sendo instanciado (GUEDES, 2012). Ou seja, um modelo UML é uma instância do metamodelo que descreve UML. Dessa forma, esse conceito disponibiliza aos usuários a liberdade de definir seus próprios modelos. Por esse motivo, a ferramenta a ser desenvolvida para composição de modelos UML, por meio de diagrama de sequência, irá utilizar do conceito de Metamodelo para tais definições.

Tal estrutura do Metamodelo é dividida em três pacotes principais: O Pacote de fundação, que define a estrutura estática da UML, através de um núcleo (relacionamentos, operações, métodos, parâmetros), tipos de dados (String, Void, etc.) e mecanismos de extensão; Pacote de Elementos comportamentais, que define a estrutura dinâmica da UML, através de um comportamento comum (sinal, operação, ação do UML), colaborações (colaboração, iteração, mensagem, classificador, função e associação do UML), os casos de uso (ator e caso de uso) e as máquinas de estado (sinais, transições, fluxo de objeto); Por fim, o pacote de gerenciamento de modelos, que define a estrutura organizacional de modelos UML, descrevendo seus pacotes e propriedades de visibilidade dos mesmos.

Através desse conceito, este trabalho irá trabalhar com o cálculo de distância entre modelos de sequência UML. Ou seja, toda a varredura que o algoritmo fizer dentre os elementos do diagrama, organizará os mesmos em Strings. Essa verificação passará desde suas lifelines (linhas do tempo) até suas mensagens (métodos), tratando e respeitando as origens de cada elemento, para que nada se perca em uma composição com outro diagrama.

Então, a partir dessas Strings geradas, se é utilizado o algoritmo para cálculo dessa distância, resultando no valor que deverá ser tratado pelo desenvolvedor para dar seguimento à composição de modelos, respeitando todas suas características e estratégias à serem utilizadas na ferramenta MoCoTo.

3 FERRAMENTA

Esta Seção tem como finalidade apresentar o algoritmo proposto e descrever como este algoritmo foi incorporado à ferramenta MoCoTo, uma ferramenta de composição de modelos UML. Para isso, essa seção é organizada da seguinte forma. A Seção 3.1 o diagrama de classes da técnica para o cálculo da distância entre diagramas de sequência e descreve como o projeto proposto é incorporado ao projeto da ferramenta MoCoTo. A Seção 3.2 mostra uma visão geral da arquitetura da ferramenta ao apresentar seu diagrama de componentes. A Seção 3.3 descreve o algoritmo para o cálculo de distância entre diagramas de sequência. Por fim, a Seção 3.4 apresentará a implementação de tal algoritmo na ferramenta MoCoTo.

3.1 Diagrama de Classes

O diagrama de classe tem como finalidade representar a estrutura de um sistema ao definir os seus conceitos e relacionamentos encontrados no domínio do problema, bem como representa decisões de projetos tomadas, através de relações, operações e atributos que servem para modelos de objetos. Na ferramenta MoCoTo, cada funcionalidade implementou o padrão *Strategy*, com o intuito de modularizar os algoritmos para realizar a composição, avaliação e comparação dos modelos de entrada. Em um geral, o diagrama de classes da ferramenta é dividido pelos pacotes: *Core*, *Merge*, *Match*, *Evaluation*, *Persistence*, *Util*.

Este trabalho, por sua vez, tem como objetivo tratar a distância entre dois diagramas de sequência UML através de sua similaridade e para isso, os pacotes afetados nessa incorporação foram o *Match* e o *Core*.

3.1.1 Pacote Match

No pacote match seus modelos de entrada são recebidos e tratados. Utilizando a classe, *DefaultMatchSequence*, os elementos dos diagramas de sequência são analisados e organizados de acordo com sua característica (lifeline ou mensagem), e assim colocados como Strings, conforme pode ser representado na figura abaixo:

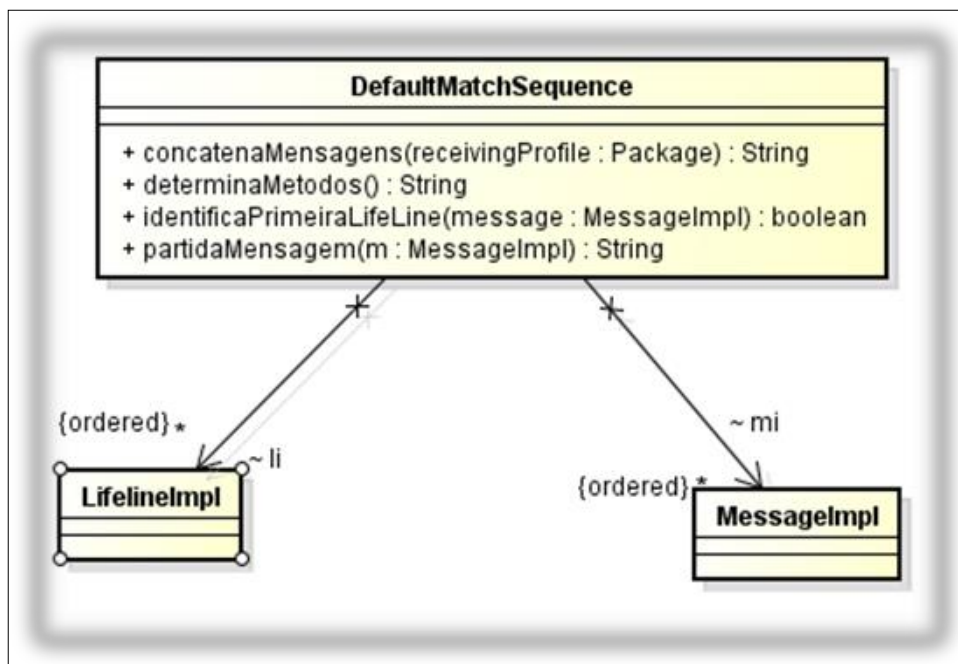


Figure 3. Diagrama de Classes – DefaultMatchSequence.

Fonte: Próprio Autor

Essa classe é caracterizada pelos seguintes métodos:

- *concatenaMensagens()*: É recebido um modelo de sequência, retornando uma String com as mensagens concatenadas;
- *determinaMetodos()*: Verifica e organiza os métodos do diagrama de sequência, indicando se as mensagens são desencadeantes (iniciadas na primeira lifeline) ou desencadeadas (executadas após as desencadeantes);
- *IdentificaPrimeiraLifeLine()*: identifica a primeira lifeline do diagrama de sequência, responsável pela origem das mensagens desencadeantes;
- *partidaMensagem()*: Identifica qual lifeline foi a origem de cada mensagem;

São utilizadas também as classes *LifelineImpl* e *MessageImpl*, que são vinculadas automaticamente ao próprio diagrama de sequência. Com elas, é possível buscar as descrições das mensagens/métodos, das lifelines, além da possibilidade de organização entre mensagens desencadeantes e desencadeadas.

3.1.2 Pacote Core

Responsável pelas chamadas das principais funções da ferramenta, é neste pacote que se é tratado o cálculo de similaridade entre os diagramas UML, através da classe *MergeEngine* conforme imagem abaixo:

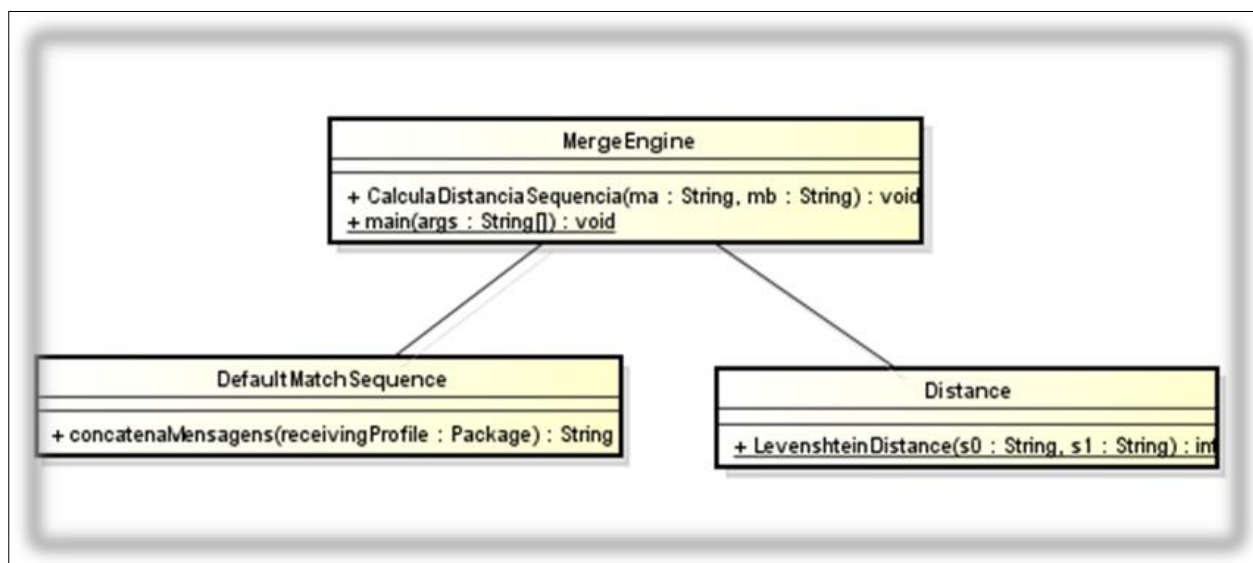


Figure 4. Diagrama de Classes – MergeEngine

Fonte: Próprio Autor

Essa classe é caracterizada pelo método *CalculaDistanciaSequencia*, onde são recebidos os modelos que irão ter seu cálculo de distância executado, retornando tal valor entre eles.

Para tal, são utilizadas também as classes *DefaultMatchSequence* (vista anteriormente), onde os elementos do diagrama são caracterizados e transformados em String e a classe *Distance* que, através do método *LevenshteinDistance* faz o tratamento das duas Strings recebidas para retornar o valor de distância entre as mesmas.

3.2 Diagrama de Componentes

O diagrama de componentes tem como principal objetivo mostrar as unidades de um software com suas interfaces bem definidas, bem como mostrar como as classes do sistema são agrupadas em tais componentes. Cada componente foi projetado para: (1) ser um módulo independente que encapsula estado e comportamento de um conjunto de elementos de executáveis, que são responsáveis pela implementação de um (ou mais) recursos; (2) apresentar comportamentos emergentes resultantes da interação de seus elementos executáveis, ou seja, uma ou mais classes que realizam as funcionalidades esperadas de recursos; e (3) tem interfaces bem definidas, incluindo as requeridas e providas. A Figura 5 ilustra o

diagrama de componentes da ferramenta MoCoTo (FARIAS et al., 2014). A ferramenta possui seis componentes, cada um sendo responsável por uma etapa dentro do processo de integração de dois modelos da UML. O algoritmo de cálculo de distância é a parte integrante destes componentes, em especial, o componente *Comparison*, que é responsável por realizar a comparação entre os dois modelos de entrada. Nesta etapa, este trabalho estende a proposta inicial da ferramenta MoCoTo ao propor um algoritmo para o cálculo de distância entre dois modelos de sequência.

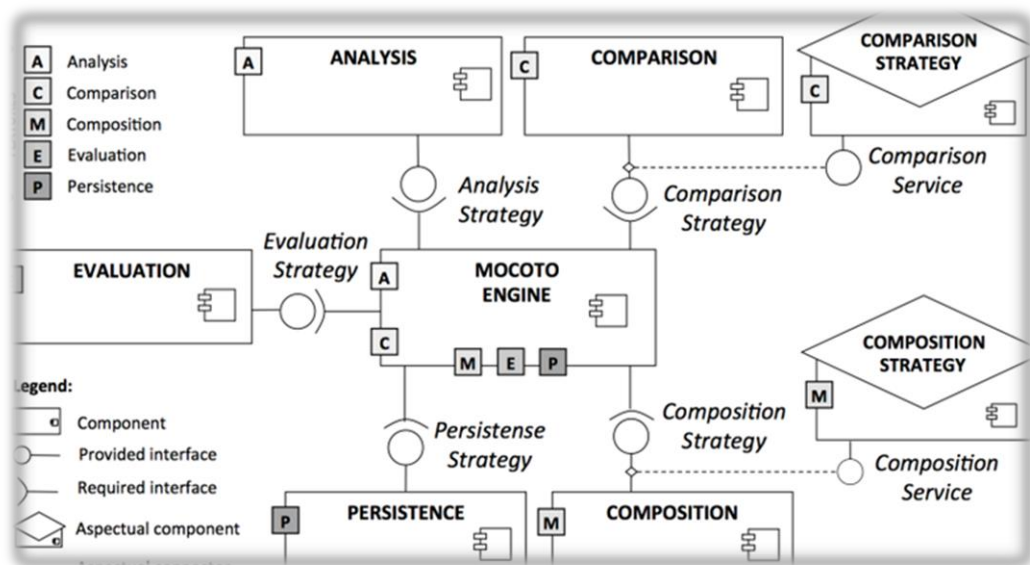


Figura 5. Diagrama de Componentes do MoCoTo SPL (FARIAS et al., 2014)

De forma complementar, o componente aspectual, *Comparison Strategy* é responsável por implementar e injetar na ferramenta MoCoTo diferentes tipos de algoritmos de comparação. Sendo assim, ele também contribui na realização da comparação entre dois modelos. Através destes componentes, o algoritmo é iniciado e a fase de comparação é realizada, resultando no valor de distância encontrado entre os modelos, que após ser tratado, poderá dar seguimento na composição dos modelos.

É importante destacar que este método de componentes de composição em unidades bem modularizadas, torna possível projetar e implementar componentes reutilizáveis.

3.3 Algoritmo de Cálculo de Distância

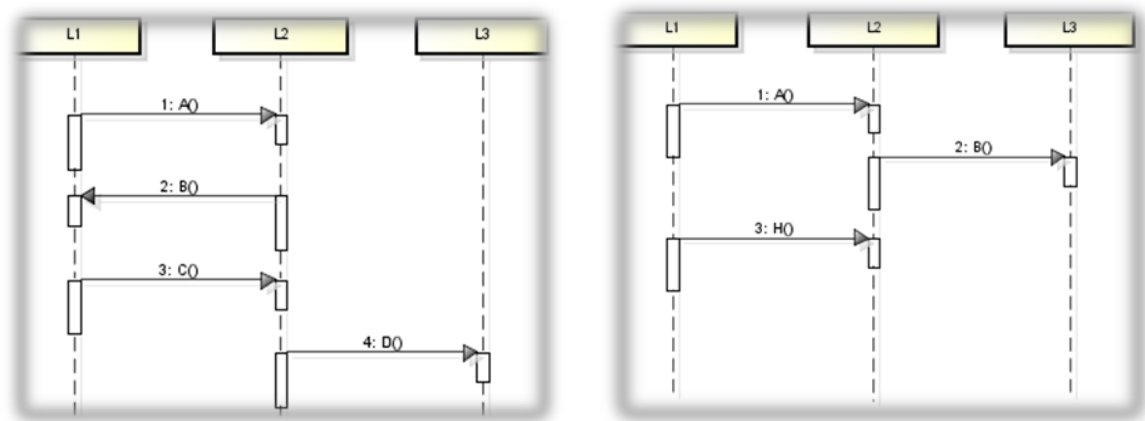
Através da organização dada dentro do desenvolvimento da ferramenta, o algoritmo para cálculo da distância entre os modelos é implementado, e seus resultados são de extrema importância para que, com um correto tratamento, a composição dos modelos em questão possa ter sua continuidade, evitando que o modelo gerado sofra de inconsistências.

O cálculo da distância entre dois diagramas de sequência, M_A e M_B , pode ser definido em dois passos. O primeiro passo consiste em converter as mensagens dos dois diagramas de sequência em uma sequência de mensagens. Dessa forma, dado dois diagramas de sequência, ao final do primeiro passo serão produzidas duas sequências de mensagens. O segundo passo consiste em calcular a distância entre as duas sequências criadas. Para isso, o algoritmo de Levenshtein (LEVENSHTEIN, 2014) será utilizado, dado que o mesmo é o algoritmo mais conhecido e amplamente usado para este propósito. Em termos práticos, o algoritmo calculará o número mínimo de operações necessárias para transformar o primeiro diagrama M_A , no segundo diagrama, M_B . As operações consideradas são: inserção, deleção e substituição de um caractere da sequência.

A Figura 6 mostra um exemplo das duas etapas citadas. Aplicando o primeiro passo no Modelo A, será feita uma varredura em todo modelo para identificar suas mensagens para transformá-las em uma sequência de caracteres. Neste caso, tem-se a seguinte sequência “L2.A, L1.B, L2.C, L3.D”, onde L2, L1 e L3 representam as linhas de vida, e A, B, C e D representam as mensagens trocadas. Desconsiderando as linhas de vida, tem-se como resultado a sequência: ABCD. A partir disso, essas mensagens são tratadas como *Strings*, dividindo em desencadeantes (método chamado como ‘principal’ para o diagrama de sequência, ou seja, utilizados na primeira linha de vida) e em desencadeados. No caso do modelo A, M_A , os principais seriam A e C. O mesmo ocorre com o modelo B, M_B da Figura 6. Através de varredura de seus elementos, a seguinte sequência é identificada: “L2.A, L3.B, L2.H”, onde L2 e L3 representam as linhas de vida, e A, B e H representam as mensagens trocadas. Desconsiderando as linhas de vida, tem-se como resultado a sequência: ABH.

Uma vez produzidas as duas sequências de caracteres dos dois diagramas da Figura 6, o segundo passo é realizado ao executar o algoritmo de Levenshtein

(LEVENSHTEIN, 2014) passando as duas sequências ABCD e ABH de caracteres produzidas.



Modelo A

Modelo B

L2.A, L1.B, L2.C, L3.D → ABCD

L2.A, L3.B, L2.H → ABH

Sequência das Mensagens do
Modelo A

Sequência das Mensagens do
Modelo B

Figura 6 – Dois diagramas de sequência para o cálculo da distância.

Os dois passos para calcular a distância entre duas sequências de mensagens dos diagramas de sequência é apresentado no pseudocódigo a seguir. Primeiramente são comparadas as linhas de vida dos diagramas, para analisar o que deverá ser colocado ou retirado, e após isso as mensagens dentre os modelos, também com o mesmo intuito de checar as operações. Feito isso, o cálculo geral da distância dentre dois modelos é realizado através do método 'LevenshteinDistance()'.

```

1  INICIA ALGORITMO (MA,MB)
2      ConcatenaMensagens (MA, MB)
3          DeterminaMetodos()
4              SE msg = Principal FAÇA
5                  principais = msg + '#'
6              SE NÃO FAÇA
7                  subs = msg
8          PRINT "MsgMA, MsgMB"
9  CALCULA DISTANCIA MÉTODOS PRINCIPAIS()
10     Distance.LevenshteinDistance(principaisA, principaisB)
11  CALCULA DISTANCIA SUB-MÉTODOS()
12     Distance.LevenshteinDistance(SubsA, SubsB)
13  PRINT "distanciaMetodosPrinc, distanciaSubMetodos"
14  CALCULA FINAL DISTANCIA
15     distancia_final = distanciaMetodosPrinc + distanciaSubMetodos
16  PRINT "distancia_final"
17  FINALIZA ALGORITMO

```

Figura 7. Algoritmo de Cálculo de Distância entre modelos UML

O algoritmo é o responsável pela busca da distância entre dois diagramas através do algoritmo de *Levenshtein* (LEVENSHTEIN, 2014), o qual é mostrado a seguir.

```

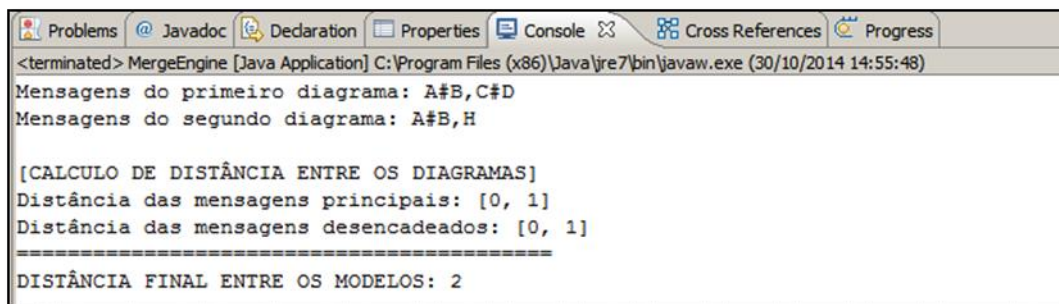
1  public class Distance {
2      public static int LevenshteinDistance (String s0, String s1) {
3          int len0 = s0.length() + 1;
4          int len1 = s1.length() + 1;
5          int[] cost = new int[len0];
6          int[] newcost = new int[len0];
7          for (int i = 0; i < len0; i++) cost[i] = i;
8          for (int j = 1; j < len1; j++) {
9              newcost[0] = j;
10             for(int i = 1; i < len0; i++) {
11                 int match = (s0.charAt(i - 1) == s1.charAt(j - 1)) ? 0 : 1;
12                 int cost_replace = cost[i - 1] + match;
13                 int cost_insert = cost[i] + 1;
14                 int cost_delete = newcost[i - 1] + 1;
15                 newcost[i] = Math.min(Math.min(cost_insert, cost_delete), cost_replace);
16             }
17             int[] swap = cost; cost = newcost; newcost = swap;
18         }
19         return cost[len0 - 1];
20     }
21 }

```

Figura 8. Algoritmo de *Levenshtein* para o cálculo da distância.

3.4 Implementação do Algoritmo

Através das regras passadas no capítulo anterior e do tratamento feito nos modelos M_A e M_B , o algoritmo é implementado e é feito o cálculo de distância dentre os dois modelos. A figura 9 mostra o resultado do cálculo de distância entre as duas sequências de mensagens dos dois diagramas de sequência.



```
<terminated> MergeEngine [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (30/10/2014 14:55:48)
Mensagens do primeiro diagrama: A#B,C#D
Mensagens do segundo diagrama: A#B,H

[CALCULO DE DISTÂNCIA ENTRE OS DIAGRAMAS]
Distância das mensagens principais: [0, 1]
Distância das mensagens desencadeados: [0, 1]
=====
DISTÂNCIA FINAL ENTRE OS MODELOS: 2
```

Figura 9. Resultado do cálculo de distância entre os modelos M_A e M_B

Conforme a avaliação identificada, o resultado da distância dentre os modelos M_A e M_B é igual a 2. Ou seja, através desse resultado o desenvolvedor poderá tratar a melhor forma de prosseguir na busca da similaridade e da composição desses dois elementos, visto que saberá qual o nível de alteração será necessário para que um modelo resultante não sofra de inconsistências e problemas.

4 AVALIAÇÃO

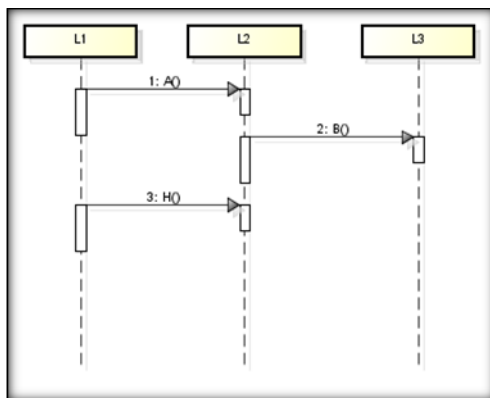
Após apresentar o algoritmo proposto e a ferramenta, esta Seção tem como objetivo avaliar tal proposta. Conforme identificado em seções anteriores, este trabalho trás o intuito de resultar no primeiro passado pela busca da similaridade, que seria a distância entre diagramas. Para tal, foi usada uma ideia menos complexo e mais dinâmica para avaliação desse algoritmo.

Para isso, alguns cenários de evolução de diagramas de sequência foram definidos e aplicados ao algoritmo proposto para avaliar a precisão do mesmo. Os cenários serão criados de uma forma que simule a evolução de um diagrama dentro de um desenvolvimento. Ou seja, será gerado um diagrama considerado como “pai” e, através dele, serão feitas mudanças (de forma manual) na sua estrutura. Serão realizados cálculos de distância a cada mudança feita, e, após isso, uma tabela de

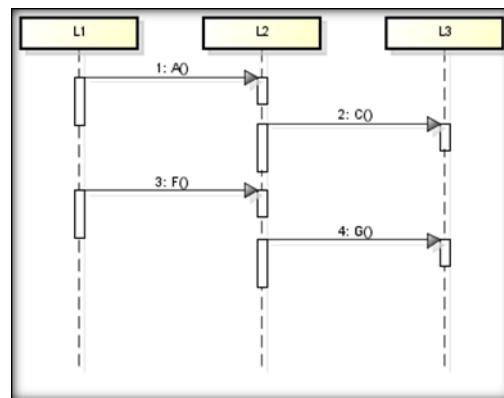
resultados será gerada para análise e averiguação do quanto tais modificações podem afetar no cálculo da distância entre dois modelos.

4.1 Cenários de Avaliação

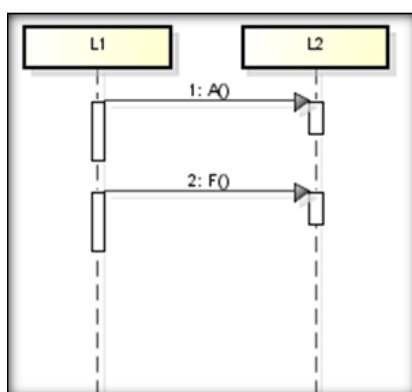
Para uma correta simulação de evolução de um diagrama dentro de um desenvolvimento, inicialmente será utilizado um diagrama M1 e, a partir deste diagrama, outros diagramas serão gerados com modificações entre mensagens e lifelines, conforme características descritas a seguir:



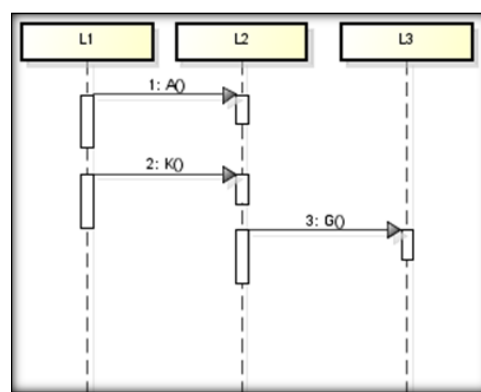
Modelo M1



Modelo M2



Modelo M3



Modelo M4

Figura 10. Cenários de avaliação propostos.

O modelo M1 contém as lifelines L1, L2 e L3, e as mensagens A, B e H. Simulando mudanças nesse diagrama dentro de um desenvolvimento, serão

retiradas as mensagens B e H e colocadas as mensagens C, F e G, gerando o modelo M2. Seguindo a evolução, um novo modelo M3 será gerado, sem a lifeline L3 e sem as mensagens C e G. Por fim, encerrando as simulações, o último modelo à ser gerado é o M4, no qual é retirada a mensagem F e acrescentada a mensagem K, além da volta da lifeline L3.

Esse tipo de avaliação tem o intuito de simular diversas mudanças que um modelo pode ter no andamento de seu desenvolvimento, visto que a partir delas que ocorre a composição de dois modelos e, se não tiver uma estratégia bem montada, e a distância (ou similaridade) entre ambos não for corretamente identificado, o novo modelo será gerado com inconsistências. Ou seja, a partir disso são simuladas algumas modificações em um diagrama pai para identificar qual será a de maior impacto em uma composição (em nível de edições de uma para outra).

4.2 Aplicação

Após definir os modelos e suas evoluções, o algoritmo é aplicado para calcular a distância dos 16 possíveis casos de cálculo de distância, considerando os quatro modelos apresentados anteriormente.

Para cada caso a ser realizado, é feita a varredura de cada modelo a ser comparado, para com seus elementos, desde suas linhas de vida (lifelines) até suas mensagens (métodos). Dessa forma, terão por si organizadas todas as mensagens desencadeantes, que partem da primeira linha de vida do modelo, e suas mensagens desencadeadas, que surgem após as mensagens consideradas “pais” operaram.

Feito isso, tendo cada elemento identificado conforme sua “categoria” são geradas Strings de acordo com a organização dada no algoritmo. Mais precisamente, conforme especificado na Seção da Ferramenta, criando grupos que iniciam com a mensagem principal e se completam com as mensagens desencadeadas.

Tendo a organização dos elementos dos diagramas já feitos e em Strings, é aplicado o algoritmo de Levenshtein (LEVENSHTEIN, 2014), calculando a distância dos mesmos para obter um resultado final. Nesse momento, são comparadas as linhas de vida e também mensagens principais de cada linha de vida, identificando

em cada um dos 16 casos aqui simulados, qual será seu grau de distância (ou similaridade), que poderá ser melhor visualizado na próxima Seção.

4.3 Avaliação

Após a aplicação do algoritmo em todos os 16 casos, conforme especificado na Seção anterior, uma tabela comparativa de resultados é gerada com o intuito de descrever a distância calculada.

Sendo assim, a Tabela 1 mostra os seguintes resultados obtidos:

	Modelo 1	Modelo 2	Modelo 3	Modelo 4
Modelo 1	0	3	2	3
Modelo 2	3	0	2	2
Modelo 3	2	2	0	2
Modelo 4	3	2	2	0

Tabela 1. Resultado da aplicação do algoritmo.

Analisando a tabela, a comparação que possui a maior distância dentre todos os casos foi encontrada nas comparações de M1xM2 e M1xM4, com o resultado de 3. Ou seja, essas evoluções devem ter uma atenção especial quanto à similaridade e consequentemente na composição de seus elementos, visto que muitas modificações serão necessárias para que o novo modelo gerado não tenha inconsistências ou não sofra com problemas de conflitos de composição.

Cabe lembrar também que, ao calcular a distância entre um mesmo diagrama, o resultado obtido é igual à zero, e essa é uma situação que pode acontecer em uma possível composição (não terem problemas na estrutura do diagrama). Tendo esse resultado zerado, evita de possíveis retrabalhos por partes dos programadores para com a estratégia de composição a partir da distância entre os modelos.

5 TRABALHOS RELACIONADOS

A busca pela similaridade entre dois elementos passará sempre, ao seu início, pelo cálculo de distância entre os mesmos, identificando as operações necessárias para a junção de ambas. Esse cálculo de distância, no geral, busca através de uma matriz de comparação entre os caracteres de Strings, esse valor que deve ser tratado para fim de dar sequência em um cálculo de similaridade.

Além do estudo de Levenshtein (LEVENSHTEIN, 2014), visto neste trabalho, outros estudos visam buscar esse resultado para fins de comparação. Como o Smith Waterman (SMITE et. Al., 1981) onde ao invés de alinhar duas Strings, como o Levenshtein, esse algoritmo alinha duas substrings. Neste algoritmo qualquer escore negativo é substituído por zero e o escore do alinhamento é o melhor escore dentre todos. Isto possibilita que nem o começo nem o fim das duas Strings precisem estar alinhado (SMITE et. Al., 1981). Outro algoritmo com uma ideia próxima a deste trabalho, é o Stochastic Model (RISTAD et. Al., 1996), no qual tem como qualidade a taxa de erro, em comparação ao algoritmo de Levenshtein, no qual fica apresentado em $\frac{1}{4}$ (um quarto). Isso, através de alinhamentos globais de Strings. Este algoritmo também é semelhante ao de Levenshtein, pois igualmente se baseia em inserções, remoções e substituições para atribuir escores a uma matriz. Escores estes que são atribuídos através de programação dinâmica (RISTAD et. Al., 1996).

O fato da busca pela distância entre diagramas de sequência UML não utilizar os algoritmos citados, surge quando os mesmos passam a utilizar uma complexidade não necessária para o foco do resultado. Apesar de serem algoritmos que, de modo geral, trazem um resultado melhor que o Levenshtein, a complexidade dos mesmos causaria um trabalho demasiado na parte que, ao menos neste trabalho, não é de grande importância.

Dessa forma, utilizando de um algoritmo com seu conceito facilitado, a utilização dos resultados de seu desenvolvimento auxilia no trabalho por parte do tratamento de dados do que está sendo comparado, que nesse caso seria por parte de dois diagramas de sequência UML.

6 CONCLUSÃO

Este capítulo demonstrou as técnicas necessárias para correta utilização do algoritmo de distância entre dois modelos UML, com o intuito de identificar da melhor

forma correta sua similaridade e fazer com que a composição de tais elementos não tenha problemas ou que seu modelo gerado não sofra com inconsistências e falhas, através de sua utilização junto à ferramenta MoCoTo. Seus resultados provaram que, tendo uma estratégia bem explícita, a utilização da distância resultante do algoritmo desenvolvido, para com identificação da similaridade, pode auxiliar na diminuição de um retrabalho diante de uma composição de diagramas, visto que o desenvolvedor saberá a qual nível de modificação estará essa composição e até que ponto poderá influenciar em um novo modelo gerado.

Apesar disso, ainda se faz necessário um estudo aprofundado quanto aos caminhos utilizados pelos desenvolvedores para tratar certa similaridade, pois mesmo com o cálculo em mãos, nem sempre o próximo tratamento a ser dado poderá resultar em uma correta composição. Uma forma seria aperfeiçoar tal algoritmo para já identificar o correto caminho a prosseguir na composição através do resultado da distância dentre os modelos e, conseqüentemente, o valor de sua similaridade. Ou seja, de acordo com o valor encontrado, certa comparação terá um tratamento especial, visto sua dificuldade no momento de uma composição. Outra forma seria o desenvolvimento de um novo algoritmo que, tendo o resultado de distância em mãos, faz o correto tratamento de similaridade para que sua composição se torne a mais rápida e fácil possível.

Porém, para que o próximo caminho seja desenvolvimento, algumas questões devem ser aplicadas e respondidas: (1) O tratamento da similaridade deve ser feito no momento da identificação da distância dentre os modelos? (2) Deve-se evitar composição de diagramas com baixo índice de similaridade? (3) Os desenvolvedores focarão nas estratégias de tratamento de uma similaridade ou apenas irão tratar suas inconsistências após um novo modelo gerado?

Sendo assim, esse trabalho representa um algoritmo para cálculo de distância entre diagramas UML, cujo intuito servirá para um aperfeiçoamento na busca da similaridade dos mesmos e de seu tratamento para uma futura composição de modelos.

An Algorithm for calculating of distance from UML sequence diagrams

Abstract: The calculation of distance between UML diagrams is important for the correct composition of models, making clear the minimum amount of issues that will have to be a model compound for another. However, the pursuit of this similarity by calculating the distance is still a big challenge for developers, since the tools available today do not present algorithms can make the analysis of this distinction between the elements of the diagrams. This paper therefore focuses on proposing an algorithm for calculating the distance between sequence diagrams to ease the identification of similarities among models. For this, the proposed algorithm was implemented in the MoCoTo composition used as a plug-in Eclipse platform, through the use of design patterns and the Levenshtein Distance Algorithm tool. The results prove the effectiveness of the algorithm through evolution scenarios and also identify the possible need for a thorough study on the use of a standard of treatment that can be done with the result of this algorithm, to assist developers in progress a composition models.

Keywords: UML. Distance between Models. Levenshtein algorithm. Composition Models.

REFERÊNCIAS

ASTAH. **Astah**. Disponível em: <http://astah.net/>. Acesso em 02.10.2014.

ECLIPSE. **Epsilon**, 2012. Em <<http://www.eclipse.org/epsilon/>>. Acesso em: 20 de junho de 2014.

FARIAS, K. **Empirical Evaluation of Effort on Composing Design Models**, PhD thesis, DI/PUC-Rio, Rio de Janeiro, Brazil. 2012.

FARIAS, K.; GARCIA, A.; WHITTLE, J. **Assessing the Impact of Aspects on Model Composition Effort**, In: 9th International Conference on Aspect-Oriented Software Development (AOSD'10), pages 73-84, Rennes and Saint-Malo, France, 2010.

FARIAS, K; GONÇALVES, L; SCHOLL, M; **Toward a Software Product Line for Model Composition Techniques**, PIPCA/UNISINOS, São Leopoldo, Brazil, 2014.

FARIAS, K., GARCIA, A., WHITTLE, J., LUCENA, C., **Analyzing the Effort of Composing Design Models of Large-Scale Software in Industrial Case Studies**, In: 16th International Conference on Model-Driven Engineering Languages and Systems (MODELS'13), pages 639-655, Miami, USA, September 2013.

FARIAS, K., GARCIA, A., LUCENA, C., **Effects of Stability on Model Composition Effort: an Exploratory Study**, Journal on Software and Systems Modeling, pages 1-22, DOI: 0.1007/s10270-012-0308-2, 2013.

FARIAS, K., GARCIA, A., WHITTLE, J., CHAVEZ, C., LUCENA, C., **Evaluating the Effort of Composing Design Models: A Controlled Experiment**, Journal on Software and Systems Modeling, pages 1-17, DOI: 10.1007/s10270-014-0408-2, 2014.

GUEDES, G. **Um Metamodelo UML para a Modelagem de Requisitos em Projetos de Sistemas MultiAgentes**. PhD thesis, INSTINFO/UFRGS, Porto Alegre, Brazil. 2012.

IBM. **Rational Unified Process**, 2008. Em <<http://www.ibm.com>>. Acesso em: 20 de junho de 2014.

LEVENSHTEIN. **The Levenshtein-Algorithm**. Disponível em: <http://www.levenshtein.net/>. Acesso em 02.10.2014.

OLIVEIRA, K. **Composição de UML Profiles**. Master's thesis, FACIN/PUCRS, Porto Alegre, Brazil. 2008.

RISTAD, E.; YANILOS P. **Learning String Edit Distance**. Research Report CS-TR-532-96, October 1996.

SILVA, M. **XSimilarity : Uma ferramenta para consultas por similaridade embutidas na linguagem XQuery**. INSTINFO/UFRGS, Porto Alegre, Brazil. 2007.

SMITE, T.; WATERMAN M. **Identification of Common Molecular Subsequences**. USA, 1980 Academic Press Inc. (London) Ltd. Reprinted from J . Mol. Biol. (1981) 147, 195-197.

UMBRELLO. **Welcome to Umbrello - The UML Modeller**. Disponível em: <https://umbrello.kde.org/>. Acesso em 02.10.2014.