An Algorithm for Distance Calculation Between UML Sequence Diagrams

J. Canal, K. Farias, and L. Gonçales

***Abstract*—The measure of distance between UML diagrams is a crucial step to support the integration of UML sequence diagrams. This distance is used for measuring the difference between two sequences messages exchanged by objects. Even though UML sequence diagram is widely used, nothing has been done to support to find matches for short sequence of messages in many longer interactions between objects. Unfortunately, calculating this distance is still a great challenge for developers given the problem at hand as semantic information is typically not included in any formal way in the models. Therefore, this work focuses on proposing an algorithm for calculating distance between sequence diagrams to enable an identification of similarity between models. For this, the proposed algorithm was implemented and incorporated into a model composition tool, namely MoCoTo. Our preliminary evaluation indicated that the algorithm was able to measure the distance between two UML sequence diagrams properly.**

***Keywords*— UML, Models Distance, Levenshtein algorithm, Model Composition, Graph Edit Distance, Sequence Diagram.**

I. INTRODUÇÃO

O CÁLCULO da distância entre dois modelos tem um papel fundamental para quantificar as diferenças entre eles. A correta identificação da distância permite, por exemplo, que técnicas de integração de modelos possam utilizar desse cálculo para realizar a integração das partes dos modelos, minimizando problemas de inconsistências [13]. O cálculo da distância entre dois modelos de sequência da UML (*Unified Modeling Language*) pode ser definido como sendo um conjunto de passos que devem ser executados para quantificar as diferenças entre as sequências de chamadas dos objetos.

No contexto de integração de modelos, calcular esta distância é fundamental para identificar quais modelos exigirão um número maior ou menor de resolução de conflitos. Por sua vez, a integração de modelos 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} [3]. Se a distância entre M_A e M_B é alta, então provavelmente o número de conflitos entre M_A e M_B tende a ser alto também, exigindo uma maior intervenção por parte dos desenvolvedores de software, os quais precisam investir esforço para produzir um modelo desejado.

Baseado em estudos empíricos, tais como [3],[11],[12], sabe-se que este esforço investido serve para resolver conflitos indesejados, visando evitar adição de inconsistências no modelo integrado. Frequentemente, os conflitos entre os elementos dos modelos de entrada, M_A e M_B, acabam sendo resolvidos de uma forma inadequada [12]. Isso ocorre principalmente pelo desconhecimento do nível de similaridade entre os elementos dos modelos. Consequentemente, um modelo composto com inconsistência é produzido, M_{CM} [3]. Quando esse modelo é gerado, a equipe de desenvolvimento precisa investir mais tempo para resolver as inconsistências, o que implica na redução da produtividade do time de desenvolvimento [4].

Com a necessidade de resolver esses conflitos, muitas ferramentas foram desenvolvidas com o objetivo de tornar clara a identificação da similaridade dos modelos, auxiliando os desenvolvedores no entendimento das operações necessárias para integrar dois modelos de software. Exemplos de ferramenta seriam a EMFCompare [1] e a ECL (*Epsilon Comparison Language*) [2].

Embora várias ferramentas e técnicas tenham sido propostas [2],[12], bem como estudos sobre integração de modelos tenham sido executados [3], o 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, os trabalhos atuais não são capazes de quantificar os quão similares (ou diferentes) os elementos do diagrama são. De fato, as ferramentas atuais não fornecem essa funcionalidade, ou por não serem capazes de identificar as similaridades, ou por não conseguirem identificar as diferenças e os conflitos.

Este trabalho, portanto, propõe um algoritmo capaz de realizar o cálculo da distância entre dois diagramas de sequência da UML. O algoritmo, que é baseado no algoritmo de *Levenshtein distance* [7], foi implementado como um plug-in da plataforma Eclipse, bem como foi integrado à ferramenta de composição de modelos MoCoTo (*Model Composition Tool* [14]). Desenvolvedores podem se beneficiar da identificação das diferenças entre as sequências das mensagens de dois diagramas de sequência da UML através do uso do algoritmo proposto. O algoritmo foi avaliado através de cenários de evolução de modelos de software para quantificar a distância entre diagramas de sequência da UML.

Este trabalho é organizado da seguinte forma. A Seção II apresenta a fundamentação teórica, a qual descreve os conceitos necessários para o entendimento do cálculo de similaridade, bem como da utilização do algoritmo com a ferramenta. A Seção III apresenta o algoritmo, bem como uma descrição de como ele foi implementado. A Seção IV

J. S. Canal, Universidade do Vale do Rio dos Sinos (UNISINOS), São Leopoldo, Brasil, joniscanal@gmail.com.

K. Farias, Universidade do Vale do Rio dos Sinos (UNISINOS), São Leopoldo, Brasil, kleinnerfarias@unisinos.com.

L. Gonçales, Universidade do Vale do Rio dos Sinos (UNISINOS), São Leopoldo, Brasil, lucianj@edu.unisinos.br

apresenta os resultados do algoritmo. E, por fim, a Seção V apresenta as conclusões e trabalhos futuros.

II. FUNDAMENTAÇÃO TEÓRICA

Esta seção descreve alguns conceitos para o entendimento do algoritmo proposto e da ferramenta à qual ele foi integrado. Para tal, a Seção II.A apresenta o conceito de diagrama de sequência. A Seção II.B apresenta o algoritmo de *Levenshtein*. A Seção II.C apresenta o conceito de integração de modelos. A Seção II.D apresenta as principais ferramentas de integração de modelos. A Seção II.E apresenta os trabalhos relacionados.

A. *Diagrama de Sequência*

Um diagrama de sequência representa visualmente a troca sequencial de mensagens entre os objetos que pertencem a um certo processo dentro do programa. A Fig. 1 apresenta um exemplo de diagrama de sequência. Neste exemplo, os objetos instanciados, que fazem parte do processo, são identificados como *lifelines* (L1, L2 e L3). As mensagens trocadas entre elas são representadas pelos rótulos "A", "B", "C" e "D". Por exemplo, a mensagem "1: A()" indica que o objeto L1 chamada um comportamento (isto é, o método L2.A()) implementado pelo objeto L2, bem como a mensagem "4: D()" representa que o objeto L2 chama o método "L3.D()". Os números de 1 a 4 indicam a ordem de execução das mensagens.

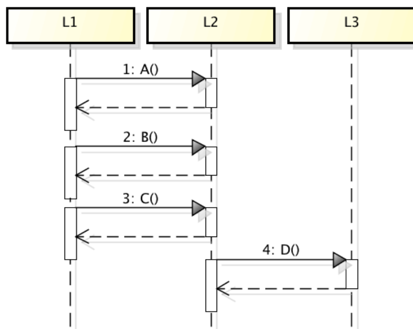

Figura 1. Exemplo de um diagrama de sequência.

B. *Levenstein Distance*

A distância de *Levenshtein* [7] é uma função que calcula a diferença com base no menor número de operações necessárias para transformar uma *String* em outra [9]. Esse algoritmo utiliza uma matriz (n + 1) × (m + 1), onde *n* e *m* representam o tamanho das duas respectivas *Strings*. Assim, ao aplicar o algoritmo de *Levenshtein* nas duas *Strings,* "classe" e "caso", pode-se identificar que é necessário remover as letras "L", e "E", e alterar o último "S" para a letra "O" para que a *String* "classe" se transforme em "caso". Como três operações foram realizadas, então a distância entre as duas *Strings* é 3.

C. *Integração de modelos*

A integração de modelos trata-se de um conjunto de atividades que devem ser executadas sobre dois modelos de entrada, M_A e M_B, com a finalidade de gerar um modelo composto desejado, M_{AB}, como anteriormente mencionado. Porém, geralmente o modelo gerado é diferente do pretendido, tendo diversas inconsistências, resultando em um modelo integrado com problemas, M_{CM}.

Sendo assim, para que essa integração possa ser melhorada, o cálculo de distância entre dois diagramas pode ser contabilizado, visando dimensionar o número de possíveis divergências que devem ser solucionados entre os modelos de entrada. Um dos problemas encontrados nas ferramentas de integração é na identificação dessa distância em diagramas de sequência.

D. *Ferramentas de Composição*

IBM RSA. A IBM RSA [6] é uma ferramenta de suporte a modelagem de software baseada na UML. Construído com base no projeto Eclipse, essa ferramenta tem um completo suporte e disponibilidade de ferramentas para modelagem UML. Apesar disso, sua utilização se limita a um processo simples, o qual não implementa algoritmos de distância para detectar as similaridades. Logo, esta responsabilidade de interpretação é delegada somente ao usuário.

Epsilon. A ferramenta Epsilon [2] fornece uma família de linguagens para comparar e compor modelos. Essa ferramenta possibilita a integração com a plataforma Eclipse, podendo utilizar também a sua grande variedade de *plug-ins* para tal plataforma. A possibilidade de reutilização do código e a alta disponibilidade de documentação e plug-ins fazem com que a Epsilon ofereça a possibilidade de uma extensão da ferramenta, através da sua adaptação para um domínio específico.

E. *Trabalhos relacionados*

Além de *Levenshtein* [7]*,* outros algoritmos visam calcular a distância entre *strings*. Em [10], os autores desenvolveram um algoritmo que alinha duas *substrings,* ao invés de alinhar duas *Strings*, como o *Levenshtein*. Neste algoritmo, qualquer valor negativo é substituído por zero e o valor do alinhamento é o melhor valor dentre todos. Isto possibilita que nem o começo nem o fim das duas *Strings* precisam estar alinhados [10]. O *Stochastic Model* [8] trata-se de outro algoritmo com propósito semelhante, o qual adota uma taxa de erro (em comparação ao algoritmo de *Levenshtein*) que fica apresentado em ¼ (um quarto). Isso é feito através de alinhamentos globais de *Strings*. Este algoritmo também é semelhante ao *Levenshtein*, pois igualmente se baseia em inserções, remoções e substituições para atribuir uma pontuação a uma matriz. Valores que são atribuídos através de programação dinâmica [8].

III. METODOLOGIA

Esta seção tem como finalidade apresentar o algoritmo proposto, bem como introduzir a ferramenta desenvolvida. Para isso, a Seção III.A descreve o algoritmo para o cálculo de distância para diagramas de sequência. A Seção III.B apresenta a implementação do algoritmo na ferramenta MoCoTo. A Seção III.C apresenta o diagrama de classes da técnica para o cálculo da distância entre diagramas de sequência. Por fim, a Seção III.D mostra a visão geral da arquitetura da ferramenta.

A. *Algoritmo de Cálculo de Distância*

O cálculo da distância entre dois diagramas de sequência da

UML, M_A e M_B, é definido em dois passos. O primeiro passo consiste em converter as mensagens de M_A e M_B em uma sequência de mensagens. Dessa forma, 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* [7] foi utilizado. Em termos práticos, o algoritmo calcula o número mínimo de operações necessárias para transformar a sequência de mensagens do primeiro diagrama M_A, na sequência de mensagens do segundo diagrama, M_B. As operações consideradas são: inserção, remoção e substituição de uma mensagem da sequência.

A Fig. 2 e Fig. 3 mostram exemplos das duas etapas citadas, respectivamente. Ao aplicar o primeiro passo em M_A, uma varredura em todo modelo é realizada para identificar suas mensagens e transformá-las em uma sequência de mensagens. Neste caso, tem-se a seguinte sequência "*L2.A, L1.B, L2.C, L3.D*", onde *L2*, *L1* e *L3* representam os objetos (*lifefines*), e *A, B, C* e *D* representam as mensagens trocadas. Desconsiderando as *lifelines*, tem-se como resultado a sequência: *ABCD*. O mesmo ocorre com o modelo M_B da Fig. 3. Através de varredura de seus elementos, a seguinte sequência é identificada: "*L2.A, L3.B, L2.H"*, onde *L2* e *L3* representam as *lifelines*, e *A, B* e *H* representam as mensagens trocadas. Desconsiderando as *lifelines*, tem-se como resultado a sequência: *ABH*. Uma vez produzidas as duas sequências de mensagens dos dois diagramas, o segundo passo é realizado aplicando o algoritmo de *Levenshtein* [7] sobre as duas sequências de caracteres *ABCD* e *ABH*.

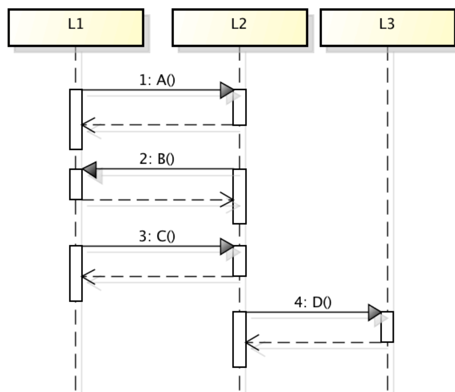

Figura 2. Modelo A, L2.A, L1.B, L2.C, L3.D → ABCD

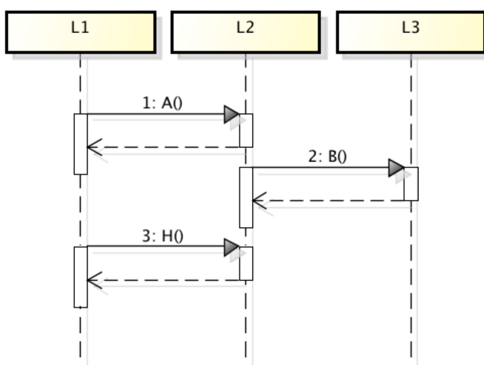

Figura 3. Modelo B, L2.A, L3.B, L2.H → ABH

Um trecho do algoritmo para calcular a distância entre as duas sequências de mensagens dos diagramas de sequência é apresentado na Fig. 4. Primeiramente, as *lifelines* dos diagramas são comparadas, para analisar o que deverá ser inserido ou retirado. Após isso, as mensagens entre os modelos também são comparadas, com o intuito de checar as operações. Feito isso, o cálculo geral da distância entre 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 4. Algoritmo de cálculo de distância entre modelos UML

O algoritmo na Fig. 5 é o responsável pelo cálculo da distância entre dois diagramas através do algoritmo de *Levenshtein* [7].

```
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 5. Algoritmo de *Levenshtein* para o cálculo da distância [6].

B. *Implementação do Algoritmo*

Através das regras passadas na seção anterior e do tratamento feito nos modelos M_A e M_B, o algoritmo foi implementado para computar o cálculo da distância entre os dois modelos de entrada. A Fig. 6 apresenta o resultado do cálculo de distância entre as duas sequências de mensagens dos dois diagramas de sequência. 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 similaridade e a composição entre esses elementos, pois saberá qual o nível de alteração que será necessário para que um modelo resultante não sofra de inconsistências e de problemas ao realizar a integração dos diagramas (Fig. 2 e Fig. 3). Nas próximas seções, o projeto da ferramenta desenvolvida será apresentado.

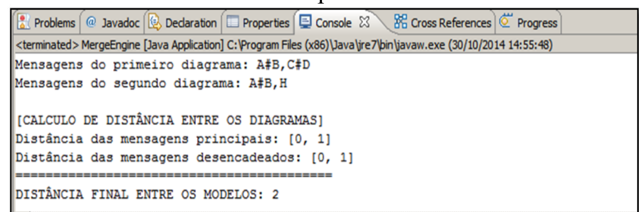

Figura 6. Resultado do cálculo de distância entre os modelos M_A e M_B.

C. *Diagrama de classes*

Esta seção apresenta as funcionalidades implementadas através de diagramas de classes. Assim, a Fig. 7 apresenta os seus conceitos e relacionamentos dentro do domínio do

problema. A função de comparar diagramas de sequência foi implementada na ferramenta MoCoTo. Esta ferramenta utiliza o padrão de projeto *Strategy* com o intuito de modularizar os algoritmos de composição, avaliação e comparação dos modelos de entrada. Este trabalho, por sua vez, se beneficiou dessa modularidade para implementar a distância entre dois diagramas de sequência UML. Assim, as funcionalidades foram estendidas através dos pontos de variabilidade fornecidos pela ferramenta. Estas extensões são descritas a seguir.

Pacote Match. Os modelos de entrada são recebidos e tratados no pacote *match*. Assim, a classe *DefaultMatchSequence* na Fig. 7 foi estendida, permitindo que os elementos dos diagramas de sequência, especificamente denominados de *lifelines*, sejam analisados e organizados de acordo com suas características.

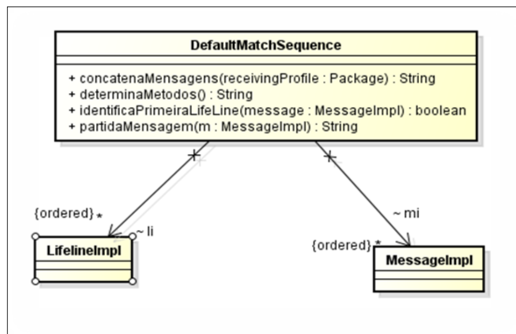

Figura 7. Diagrama de Classe *DefaultMatchSequence*.

Essa classe é caracterizada pelos seguintes métodos:

- *concatenaMensagens():* recebe um modelo de sequência, retornando uma *String* com as mensagens concatenadas;
- *determinaMetodos():* verifica e organiza os métodos do diagrama de sequência e indica 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 *MessagemImpl*, as quais são vinculadas 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.

Pacote Core. Responsável pelas chamadas das principais funções da ferramenta, é neste pacote que é tratado o cálculo de distância entre os diagramas UML, utilizando a classe *MergeEngine* conforme a Fig. 8. Essa classe contém o método *CalculaDistanciaSequencia* que calcula a distância entre os modelos de entrada. Também são utilizadas as classes *DefaultMatchSequence,* que foi estendida com o método *concatenaMensagens*, onde os elementos do diagrama são caracterizados e transformados em *Strings,* e a classe *Distance* que, através do método *LevenshteinDistance* faz o tratamento das *Strings* de entrada para retornar o valor de distância entre elas.

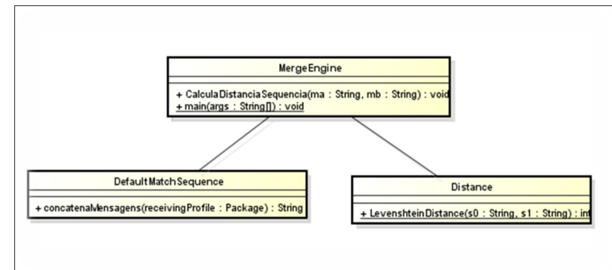

Figura 8. Diagrama de Classes.

D. *Diagrama de Componentes.*

A Fig. 9 ilustra o diagrama de componentes da ferramenta MoCoTo [3]. Os principais componentes da ferramenta são especificados por interfaces bem definidas, os quais estão associados ao núcleo central da ferramenta, MoCoTo *Engine*. Além disso, os principais módulos são *Analysis*, *Comparison*, *Composition*, *Evaluation* e *Persistence*. Cada módulo é independente e contém funcionalidades encapsuladas relacionadas à análise, comparação, composição, avaliação e persistência de modelos, respectivamente. Esses componentes são responsáveis por uma etapa dentro do processo de integração entre modelos de software.

O algoritmo de cálculo de distância é parte integrante do componente *Comparison*, que é responsável por realizar a comparação entre os dois modelos de entrada. O componente *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, a comparação é realizada, resultando no valor de distância entre os modelos, que após ser tratado, poderá dar seguimento a integração dos modelos.

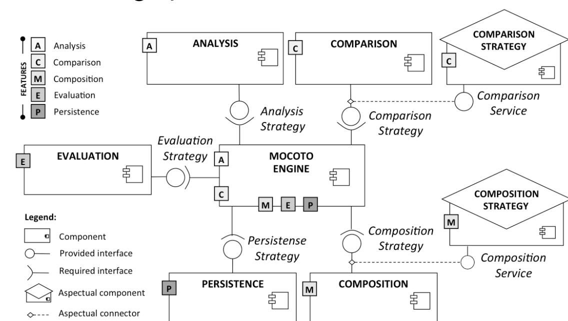

Figura 9. Diagrama de Componentes do MoCoTo [14].

IV. RESULTADOS

Alguns cenários de evolução [15] de diagramas de sequência foram definidos para avaliar a precisão do algoritmo proposto. Os cenários de evolução de modelos formulados visam identificar a distância entre um modelo modificado em relação a um modelo base. Assim, um diagrama M_B é obtido a partir das alterações de um modelo base, M_A. Desse modo, busca-se quantificar a distância entre M_A e M_B.

A comparação entre M_A e M_B ocorre com o objetivo de acomodar as alterações em M_A, sem inconsistências, identificando corretamente os elementos que são equivalentes, os que devem ser removidos, e adicionados ao modelo base, M_A. Uma tabela de resultados foi gerada e analisada.

A. *Cenários de Avaliação.*

Para uma correta simulação de evolução de um diagrama [14] dentro de um desenvolvimento, foi utilizado um diagrama *M1* e, a partir deste diagrama, outros diagramas foram gerados com modificações entre mensagens e *lifelines*, conforme características da Fig. 10, Fig 11, Fig. 12 e Fig. 13.

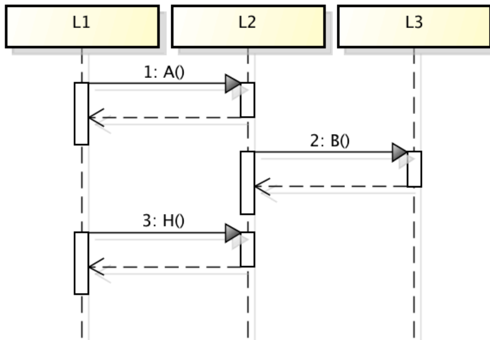

Figura 10. Modelo M1.

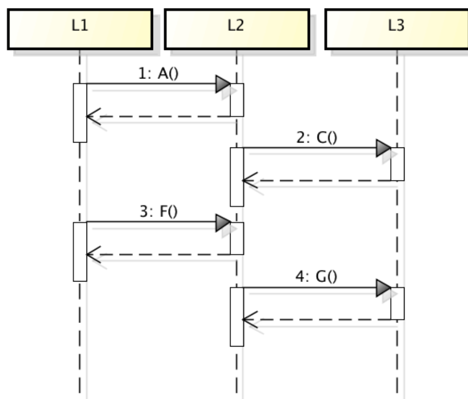

Figura 11. Modelo M2.

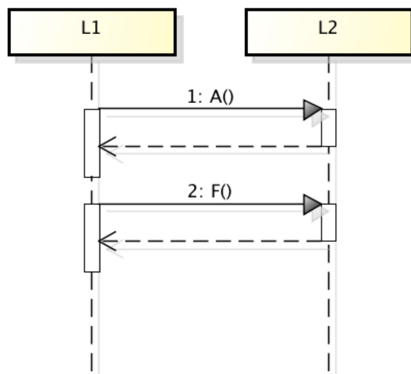

Figura 12. Modelo M3.

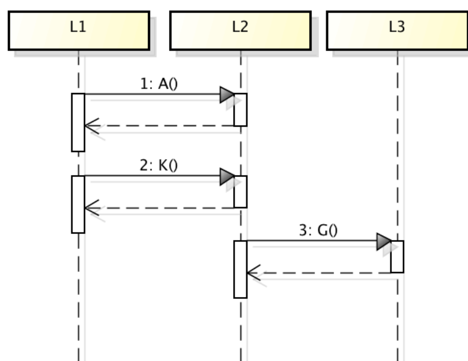

Figura 13. Modelo M4.

O modelo *M1* contém as *lifelines L1, L2* e *L3*, e as mensagens *A, B* e *H*. Implementando evoluções nesse diagrama, são removidas as mensagens *B* e *H* e colocadas as mensagens *C, F e G*, gerando o modelo *M2* (Fig. 11). Seguindo a evolução, um novo modelo *M3* (Fig. 12) será gerado, sem a *lifeline* L3 e sem as mensagens *C* e *G*. Por fim, encerrando as simulações, o último modelo a ser gerado é o *M4* (Fig. 13), no qual é retirada a mensagem *F* e acrescentada a mensagem *K*, além da volta da *lifeline L3*.

Esse tipo de avaliação visa simular as mudanças que um modelo sofre durante sua evolução, onde geralmente inconsistências são geradas.

B. *Aplicando o algoritmo*

Após definir os modelos e suas evoluções, o algoritmo é aplicado para calcular a distância entre os modelos citados anteriormente. Para cada caso, é feita a varredura de cada modelo a ser comparado, capturando cada um de seus elementos, desde suas *lifelines* (objetos) até suas mensagens (métodos). Dessa forma, todas as mensagens desencadeantes e suas mensagens desencadeadas serão organizadas. Feito isso, tendo cada elemento identificado conforme sua "categoria", são gerados *Strings* de acordo com a organização no algoritmo. Mais precisamente, conforme especificado na Seção III, criando grupos que iniciam com a mensagem principal e se completam com as mensagens desencadeadas.

Tendo a organização dos elementos dos diagramas já transformados em *Strings*, o algoritmo de *Levenshtein* [7] é aplicado, calculando a distância dos mesmos para obter um resultado final. Nesse momento, as *lifelines* também mensagens principais de cada linha de vida são comparadas com a finalidade de obter o grau de distância (ou similaridade) entre os 16 casos de comparação. Essa operação é melhor visualizada na próxima Seção.

C. *Resultados obtidos*

Após a aplicação do algoritmo em todos os 16 casos, conforme especificado na Seção anterior, uma tabela comparativa dos resultados é gerada com o intuito de descrever a distância calculada. Sendo assim, a Tab. 1 apresenta os resultados obtidos no experimento.

TABELA 1. RESULTADO DA APLICAÇÃO DO ALGORITMO.

	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**

Analisando a tabela, os modelos que possuem a maior distância dentre todos os casos seriam os M1xM2, M1xM4, com o resultado de 3. Ou seja, essas evoluções devem ter uma atenção especial quanto à similaridade e, consequentemente, na integração 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 integração.

V. CONCLUSÃO

Este artigo apresentou um algoritmo para o cálculo de distância entre dois diagramas de sequência da UML. Cenários de aplicação foram elaborados com o objetivo de mostrar a utilidade e efetividade da abordagem proposta. Essa melhoria permite, por exemplo, reduzir o esforço que desenvolvedores investem para integrar modelos, o qual é visto como um dos problemas ainda em aberto na área [12],[13]. O algoritmo foi implementado e incorporado à ferramenta MoCoTo.

O algoritmo foi avaliado em um conjunto de cenários e os resultados obtidos apontam que o algoritmo foi útil para identificar a distância entre modelos de diagramas de sequência da UML. O cálculo da distância pode auxiliar na diminuição de retrabalho diante de uma integração de diagramas com conflitos, visto que o desenvolvedor poderá dimensionar a quantidade de conflitos, bem como entender se a distância entre os diagramas pode influenciar positivamente ou negativamente na produção do modelo desejado.

Apesar do algoritmo ter apresentado resultados favoráveis durante a avaliação, futuramente pretende-se refinar o cálculo da distância através de novas estratégias de avaliação de diagramas de sequência, tais como aspectos semânticos e estruturais. Além disso, serão executados estudos experimentais com objetivo de identificar fatores que podem afetar a integração de diagramas de sequência, tais como o esforço investido para integrar modelos pelos desenvolvedores.

Assim, as seguintes questões de pesquisa poderiam ser investigadas em trabalhos futuros: (1) o cálculo da similaridade pode ser feito no momento da identificação da distância entre os modelos?; (2) deve-se evitar composição de diagramas com baixo índice de similaridade?; e (3) os desenvolvedores focarão nas estratégias de tratamento de uma similaridade, ou apenas irão tratar suas inconsistências após um novo modelo gerado? Por fim, este trabalho pode ser visto como um primeiro passo para o desenvolvimento de uma agenda de pesquisa focada no desenvolvimento de novas técnicas para dar suporte ao cálculo de distância de diagramas comportamentais da UML.

AGRADECIMENTOS

Este trabalho contou com o suporte do projeto universal – CNPq (480468/2013-3).

REFERÊNCIAS BIBLIOGRÁFICAS

[1] EMFCompare. A tool for comparing and merge EMF models. URL: https://www.eclipse.org/emf/compare/.

[2] EPSILON. *Epsilon Language*, URL: http://www.eclipse.org/epsilon/.

[3] K. FARIAS, A. GARCIA, J. WHITTLE, C. LUCENA, “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.

[4] R. M. VANALLE, G. L. BAPTISTA, J. A. SALLES. “A Software Development Process Model Integrated to a Performance Measurement System”. IEEE Latin America Transactions, v. 13, n. 3, p. 739-745, 2015.

[5] T. A. GILLEANES, G. GUEDES. “Um Metamodelo UML para a Modelagem de Requisitos em Projetos de Sistemas Multi Agentes”. Tese de Doutorado, INSTINFO/UFRGS, Porto Alegre, Brasil. 2012.

[6] IBM. “Rational Unified Process”. URL: http://www.ibm.com.

[7] LEVENSHTEIN. “The Levenshtein-Algorithm”. URL: http://www.levenshtein.net/.

[8] E. S. RISTAD., P. R NYANILOS. “Learning String Edit Distance”. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 20, num. 5, pp. 522-532, 1998.

[9] M. S. V. SILVA, E. N. BORGES, R. GALANTE. “XSimilarity: Uma ferramenta para consultas por similaridade” IV Escola Regional de Bancos de Dados – ERBD, 2008.

[10] T. SMITE, M. WATERMAN. “Identification of Common Molecular Subsequences”. Academic Press Inc. (London) Ltd. Reprinted from J. Mol. Biol., pp. 195-197, 1981.

[11] K. FARIAS, “Empirical Evaluation of Effort on Composing Design Models”, PhD Thesis, Department of Informatics, PUC-Rio, Rio de Janeiro, Brazil, 2012.

[12] M. L. ROSA, M. DUMAS, R. UBA, R. DIJKMAN, “Business process model merging: An approach to business process consolidation”, ACM Transactions on Software Engineering and Methodology. Vol. 22, no. 2, pp. 11-42, 2013.

[13] K. FARIAS, A. GARCIA, J. WHITTLE, C. CHAVEZ, C. LUCENA, “Evaluating the effort of composing design models: a controlled experiment”. Software & Systems Modeling, vol. 14, no. 4, pp, 1349-1365, 2015.

[14] K. FARIAS, L. GONÇALES, M. SCHOLL, T. C. OLIVEIRA, M. VERONEZ. “Toward an Architecture for Model Composition Techniques”, In: 27th International Conference on Software Engineering and Knowledge Engineering, pp. 656-659. 2015.

[15] N. BARBOSA, K. HIRAMA. “Assessment of Software Maintainability Evolution Using C&K Metrics”. IEEE Latin America Transactions, vol. 11, no. 5, pp. 1232-1237, 2013.

Joni Canal is an Implementation Analyst at the Digital Commerce Group (DCG). He completed his studies in Analysis and Development of Systems at the University of Vale do Rio dos Sinos (UNISINOS) in 2014. His current interests are linked to the in-depth study of agile methods in project management, and improvements on PMP (Project Management Professional).

Kleinner Farias is an Assistant Professor in the Interdisciplinary Postgraduate Program in Applied Computing at the University of Vale dos Rio dos Sinos (Unisinos). He is an associate member of the OPUS Researcher Group at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil. He received his Ph.D. in Computer Science from PUC-Rio in 2012. He received his Master’s degree in Computer Science from the Pontifical Catholic University of Rio Grande do Sul (PUC-RS) in 2008. He completed his undergraduate studies in Computer Science at the Federal University of Alagoas and in Information Technology at the Federal Institute of Alagoas in 2006. His current research interests include software modeling, empirical evaluation of model composition techniques, software metrics, and neuroscience applied to software engineering.

Lucian Gonçales is a PhD Student in the Graduate Program on Applied Computing (PPGCA) at the University of Vale dos Rio dos Sinos (Unisinos). His current research is about software modeling, neuroscience, and empirical evaluation of model composition techniques.