Software para geração automatizada de casos de teste funcionais utilizando diagramas de sequência em UML

Fernanda Ressler Feiten[1] | Francisco Assis Moreira do Nascimento[2]

Resumo

O artigo apresenta o funcionamento e resultados obtidos com o desenvolvimento do *software* UML2UMLTesting, que permite a geração de casos de teste de forma automatizada utilizando diagramas de sequência em UML criados durante a especificação do sistema. Com isso, problemas decorrentes de má interpretação da especificação, da omissão de dados causados devido à criação manual dos testes serão evitados, além de reduzir o tempo do ciclo de testes e permitir que esse seja iniciado juntamente ao ciclo inicial de desenvolvimento do sistema. Os casos de teste gerados pelo UML2UMLTesting podem ser utilizados em ferramentas de automação de testes, assim como base para a execução dos testes de forma manual pelo testador.

Palavras-chave: Casos de teste. Teste baseado em modelos. MDA. UML. ATL.

Abstract

This article presents the operation and the results obtained from the development of UML2UML Testing, a software that allows automated generation of test cases with UML sequence diagrams that are created according to system requirement specifications. Thus, problems related to misinterpreted specification and omission of data caused by manual test cases will be avoided. In addition, there will be a time cycle reduction in software testing that now will be initiated in the first stage of a system development life cycle. Test cases generated by UML2UML Testing can be applied to test automation tools and serve as a base for testers in manual testing.

Keywords: *Test cases. Model-based testing. MDA. UML. ATL.*

[1] Graduada em Sistemas de Informação pelas Faculdades Integradas de Taquara - Faccat - Taquara/RS. fernandarf@faccat.br
[2] Professor das Faculdades Integradas de Taquara - Faccat - Taquara/RS. Orientador do trabalho.
assis@faccat.br - http://lattes.cnpq.br/4204348119100399

1 Introdução

Para Pressman (2006), teste de *software* pode ser descrito como um conjunto de atividades voltadas para a verificação e validação de um *software*, podendo essas ser planejadas antecipadamente e conduzidas de forma sistemática.

De acordo com Perry (2006), com os casos de teste sendo criados no início do ciclo de desenvolvimento, é possível garantir uma versão do sistema em desenvolvimento para avaliação do cliente mais cedo, antes desse estar completo.

A importância da geração automatizada dos casos de teste é apresentada em Lindlar, Windisch, Wegener (2010), os quais afirmam que algumas atividades realizadas manualmente, como a projeção dos casos de teste, a seleção dos dados e a avaliação dos testes, exigem e consomem uma quantidade significativa de tempo. Essas atividades poderiam ser efetuadas de forma automatizada, garantindo uma melhor qualidade, pois evitariam erros causados pelo testador, além de possibilitarem que os testes sejam realizados com maior frequência e com maior antecedência.

Com o objetivo de evitar erros causados durante a criação dos testes, foi desenvolvido um sistema capaz de gerar os casos de teste de forma automatizada que faz uso dos diagramas de sequência em UML criados durante a especificação do sistema. Para isso, foram utilizados conceitos de testes baseados na especificação para o desenvolvimento do *software*.

Além disso, a ferramenta de geração de casos de teste funcionais UML2UMLTesting visa permitir que os testes sejam criados o mais cedo possível, sem a necessidade da espera de uma entrega funcional do sistema, evitando erros causados por uma má interpretação da especificação.

Como os testes funcionais são voltados para a validação das funcionalidades do sistema, a proposta do UML2UMLTesting é fazer uso das mensagens trocadas entre o ator (usuário) e o sistema para a criação dos testes.

O artigo apresenta, na seção 2, o referencial teórico e, na seção 3, a metodologia adotada. Na seção 4, são apresentados experimentos que foram realizados e os resultados obtidos a partir desses. Na seção 5, são apresentadas as conclusões.

2 Referencial teórico

2.1 Teste de *software*

O teste é considerado um elemento importante para a garantia de qualidade de um *software*, pois é representado como uma revisão final de todo o processo de desenvolvimento do sistema, englobando a especificação, o projeto e a até a geração do código (PRESSMAN, 2006).

O teste de *software*, para Rios e Moreira Filho (2006), é definido como um processo no qual o comportamento do *software* é avaliado de acordo com o que foi especificado, além de considerar a execução dos testes como um tipo de validação para o

software. Burnstein (2002) complementa a visão de testes de *software* de Rios e Moreira Filho (2006), considerando que esse processo deve ser utilizado para revelar defeitos no *software* e também garantir que tenha sido alcançado um determinado nível de qualidade.

Os testes têm como objetivo encontrar falhas antes de o sistema ser entregue ao cliente, uma vez que, quanto mais tarde essas falhas são encontradas, mais caro é o custo para a correção dessas (RIOS; MOREIRA FILHO, 2006).

Testes funcionais pertencem à abordagem caixa-preta (*Black-box*), pois são planejados utilizando a especificação e não o código do sistema (RIOS; MOREIRA FILHO, 2006). Esses têm como objetivo verificar as funcionalidades da aplicação, sem se preocupar com a lógica e métodos utilizados no sistema em testes (RIOS; MOREIRA FILHO, 2006).

Esse tipo de teste é utilizado para garantir que o comportamento do sistema esteja de acordo com a especificação de requisitos, e o foco desse é nas entradas e saídas adequadas para cada função, uma vez que todas as funcionalidades do sistema devem ser testadas (BURNSTEIN, 2002).

Casos de teste consistem em conjuntos de testes a serem executados, que visam à garantia de uma maior probabilidade na detecção de erros no sistema a ser testado (PRESSMAN, 2006). Conforme descreve Perry (2006), um caso de teste é definido por um conjunto de entradas de teste, condições de execução e os resultados esperados para atingir um objetivo de um teste em particular.

Teste baseado em modelo consiste em um teste gerado a partir de modelo que descreve o comportamento esperado para o *software* ou parte dele, utilizando para tanto como referência o método de teste caixa preta (JACKY *et al.*, 2008). Ainda conforme os autores (2008), essa prática é utilizada para a geração automática de casos de teste, em que é utilizado um modelo formal e funcional do sistema em testes (SUT[3]).

De acordo com Reza e Lande (2010), teste baseado em modelo pode ser descrito como uma técnica de teste, da qual é possível, a partir de requisitos e de modelos comportamentais do sistema, gerar de forma automática casos de teste.

Como o teste baseado em modelo é gerado a partir da especificação do sistema, é possível iniciar o processo de testes logo após os requisitos estarem definidos, sem a necessidade de espera do término do processo de desenvolvimento. Além disso, outro benefício adquirido com o teste baseado em modelos é a redução do custo para a geração dos testes, já que com o uso dessa técnica se adquire uma redução de tempo gasto com a criação desses durante o ciclo de teste (REZA; LANDE, 2010).

2.2 UML (*Unified Modeling Language*)

Conforme descrito em Alhir (2002), a UML é uma linguagem para a especificação, visualização, construção e documentação de artefatos do processo de um sistema. Essa consiste em um padrão para a criação de modelos, sendo flexível e independente de linguagens de programação (PENDER, 2002; LIMA, 2011).

A UML foi criada como um padrão de notações gráficas pela OMG[4] em 1997 e

[3] SUT: *System under test.*
[4] OMG: *Object Management Group.*

tem sido utilizada até hoje, encontrando-se, atualmente, na segunda versão (LARMAN, 2001).

Essa é composta por diagramas, que descrevem o sistema através de modelos, sendo utilizada para projetar sistemas orientados a objetos. Esses modelos são formados por um conjunto de ideias que incluem informações necessárias para o entendimento e eliminam qualquer tipo de informação irrelevante ou que possa vir a dificultar o entendimento sobre o sistema (ALHIR, 2002).

2.3 XMI (*XML Metadata Interchange*)

Segundo Grose, Doney e Brodsky (2002), XMI é um padrão da OMG utilizado pela MDA (*Model Driven Architecture*), que permite gerar uma representação em XML (*Extensible Markup Language*) para modelos UML, tornando a criação de modelos em UML mais prática e evitando erros ao fazer isso manualmente. Ou seja, o XMI especifica como devem ser criados os esquemas de XML partindo de modelos, sendo utilizado para mapear modelos UML dentro de um XML (PENDER, 2002).

O XMI tem como finalidade permitir uma troca de dados sobre modelos UML entre ferramentas de modelagens diferentes, garantindo, assim, uma maior compatibilidade entre plataformas e linguagens distintas (PAES, 2009).

Alguns *softwares* para geração de diagramas UML possuem suas próprias DTD[5]. Com isso, ao exportar esses modelos para o formato XMI, esses são criados com base na DTD da ferramenta em uso (PENDER, 2002).

2.4 MDA (*Model Driven Architecture*)

MDA, arquitetura orientada por modelos, consiste em um *framework* definido pela OMG em 2001 para o desenvolvimento de *software*, no qual modelos são fundamentais para esse processo, tornando o desenvolvimento dirigido pela modelagem do sistema (KLEPPE; WARMER; BAST, 2003). Ao contrário de outros *frameworks*, como o CORBA (*Common Object Request Broker Architecture*), a MDA não é feito para implementação de sistemas distribuídos, mas para uma abordagem de desenvolvimento de *software* com o uso de modelos (OMG, 2003).

A MDA faz uso de linguagens de modelagem baseadas em padrões como linguagem de desenvolvimento formal, as quais são diferentes das linguagens tradicionais, já que se obtém uma melhora na produtividade, qualidade e na perspectiva de longevidade (FRANKEL, 2003).

O objetivo da MDA não é gerar mudanças radicais na forma que se melhora o desenvolvimento de *software* atual, mas, sim, consolidar essas formas que ajudam a melhorar a produção de *software* (FRANKEL, 2003). Ainda conforme Souza e Araújo (2009), a proposta da MDA é tornar o desenvolvimento mais independente, separando plataforma, tecnologia e lógica de negócio, permitindo que um não interfira no outro,

[5] DTD: *Document Type Definition.*

adquirindo, assim, plataformas que não afetem as existentes e modificar a lógica de negócio sem necessidade de preocupação com a tecnologia.

Para o uso da MDA no processo de desenvolvimento de *software*, é necessário o uso de modelos definidos por essa arquitetura, sendo o primeiro deles conhecido como PIM[6], modelo de mais alto nível de abstração, usado para representar as regras de negócio (KLEPPE; WARMER; BAST, 2003). Esse modelo então é transformado em um segundo modelo, conhecido como PSM[7], o qual especifica o sistema em termos de desenvolvimento, demonstrando detalhes específicos para uma plataforma em particular, sendo considerado de mais baixo nível e utilizado pelos desenvolvedores do sistema (KLEPPE; WARMER; BAST, 2003). A partir do PSM criado, são gerados os códigos fonte para o sistema (SOUZA; ARAÚJO, 2009).

O modelo independente de plataforma pode ser utilizado para a geração de mais de um modelo para plataforma específica, em que cada um possuirá uma plataforma de tecnologia específica, sendo essa transformação, entre um modelo e outro, considerada por Kleppe *et al.* (2003) a mais complexa da MDA.

Um metamodelo é um modelo que descreve outro modelo, com nível diferente de abstração (FRANKEL, 2003). Segundo Mellor *et al.* (2005), um metamodelo é um modelo da linguagem de modelagem e, a partir desse, são definidas a estrutura, restrições e semântica para um ou mais modelos. Esses podem ser considerados como facilitadores na comunicação entre modelos (MELLOR *et al.*, 2005).

O metamodelo é geralmente representado por um diagrama de classe, podendo algumas vezes ser representado por um diagrama de entidade relacionamento, que define os conceitos da linguagem (RECH; BUNSE, 2009). Esses são descritos por uma linguagem própria, que passa a ser chamada de metalinguagem, sendo um exemplo dessas o *ecore*, da plataforma Eclipse.

O metamodelo UML, que define o modelo UML padrão, é especificado por meio do padrão MOF[8], esse sendo responsável pela descrição dos aspectos comportamentais e estruturais de um modelo em UML (MELLOR *et al.*, 2005). As representações gráficas desses modelos não são definidas no metamodelo MOF, apenas é definido como os aspectos contidos nesse serão acessados, como, por exemplo, pelo XMI (MELLOR *et al.*, 2005).

2.5 ATL (*Atlas Transformation Language*)

ATL é uma linguagem de transformação de modelos muito utilizada em MDA, que faz uso de elementos de um modelo inicial, que está em conformidade com um determinado metamodelo, para gerar um novo modelo que deve estar em conformidade com um outro dado metamodelo, ou seja, produz modelos partindo de outros modelos (PAES, 2009).

A ATL é uma linguagem híbrida e imperativa, baseada em OCL[9], desenvolvida pelo grupo de pesquisa INRIA & LINA para a plataforma Eclipse como concorrente de padrões

[6] PIM: *Platform Independent Model*, modelo independente de plataforma (PAES, 2009).
[7] PSM: *Platform Specific Model*, modelo específico de plataforma (PAES, 2009).
[8] MOF: *MetaObject Facility*, recurso padronizado e especificado pela OMG.
[9] OCL: *Object Constraint Language*, linguagem de especificação formal, usada para criar restrições sobre objetos.

criados pela OMG, como o MOF e QVT RFP[10] (INRIA ATLAS, 2006). Para gerar um novo modelo utilizando ATL, faz-se uso de regras que compõem a linguagem e que permitem que sejam definidos quais os elementos do modelo de origem serão usados para criar e inicializar os elementos do novo modelo (INRIA ATLAS, 2006).

2.6 Trabalhos correlatos

A geração de casos de teste a partir da técnica MBT já vem sendo utilizada há algum tempo, para diversos fins e utilizando diferentes meios. Dentre esses, é possível citar o trabalho de Lamancha *et al.* (2009), no qual é apresentada uma proposta de *software* para testes automatizados baseados em modelos, utilizando o UML-TP[11].

No trabalho de Lamancha *et al.* (2009), os testes são gerados a partir de diagramas de classes e sequência, que são transformados através da linguagem de transformação entre modelos QVT, em novos diagramas de sequência e classe conforme as especificações do UML-TP, utilizado como metamodelo (LAMANCHA *et al.*, 2009). A escolha pelo uso de diagramas de sequência se deve ao fato de esses demonstrarem o comportamento do sistema para um determinado caso, podendo ser utilizado para representar o comportamento de casos de teste (LAMANCHA *et al.*, 2009).

Conforme Lamancha *et al.* (2009), alguns problemas foram encontrados com o uso da linguagem QVT, que, por ser um padrão OMG, não possui total suporte na plataforma Eclipse. O uso do UML-TP como metamodelo também foi apresentado como uma dificuldade encontrada, pois esse não possui uma versão oficial como modelo UML baseado no EMF (*Eclipse Modeling Framework*) (LAMANCHA *et al.*, 2009).

No trabalho proposto por Cartaxo (2006), é apresentado o *software* LTS-BT (*Labeled Transition System-Based Testing)* para a geração de casos de teste funcionais para aplicações de celulares, fazendo uso de diagramas de sequência como entrada. Esses modelos de sequência são gerados em UML durante o desenvolvimento do sistema que será testado, precisando ser convertido para LTS[12], pois, conforme Cartaxo (2006), os modelos em UML apresentam problemas de notação para fluxos alternativos.

De acordo com Cartaxo (2006), além de o sistema depender de modelos em LTS, para gerar a transformação dos modelos UML em LTS, é necessária a utilização dos formatos *Rose* ou *Rose RT,* criados através das ferramentas *IBM Rational Rose e IBM Rational Rose Real Time*, o que dificulta a utilização do *software*, por serem necessárias licenças para o uso dessas ferramentas.

Tendo como base os trabalhos citados anteriormente, foi criado um *software*, utilizando conceitos da MDA, capaz de gerar casos de teste funcionais em formato de diagrama de atividade textual a partir de diagramas de sequência criados durante a especificação do sistema em testes. Como forma de evitar problemas encontrados pelos

---

[10] QVT RFP: QVT (*Query/View/Transformation*) padrão OMG para linguagem de transformação de modelos, à qual foi solicitada uma proposta (RFP) em MOF de padrão de compatiblidade com o pacote de recomendações da MDA.

[11] UML-TP: UML *testing profile*, padrão criado pela OMG para definir uma linguagem para especificação, visualização e análise voltada para testes (LAMANCHA *et al.*, 2009).

[12] LTS: *Labeled Transition System*, descreve integralmente todos os possíveis comportamentos do sistema (CARTAXO, 2006).

autores citados, foram utilizadas ferramentas e linguagens de transformação entre modelos diferentes das propostas nos trabalhos de Lamancha *et al.* (2009) e Cartaxo (2006).

3 Metodologia

A partir da análise dos problemas apontados em trabalhos correlatos por alguns autores para a criação manual de testes, foi desenvolvido o *software* UML2UMLTesting, no qual o ciclo de desenvolvimento e o de testes aconteceram em paralelo, possibilitando a criação de forma automatizada de casos de teste a partir de diagramas de sequência obtidos da especificação do sistema.

Para o desenvolvimento, foram utilizadas as linguagens Java e ATL, linguagem própria para a transformação entre modelos, realizados experimentos com diagramas de sequência criados e exportados através da ferramenta Magic Draw, a qual possui suporte a exportação de diagramas em UML2, para que fosse possível validar as funcionalidades até então desenvolvidas.

Com o objetivo de permitir que a ferramenta possa ser utilizada durante o ciclo de testes pelo testador, foi desenvolvida uma interface para o sistema, através da linguagem Java, permitindo a seleção do arquivo de entrada, diagrama de sequência e o local onde o arquivo com os casos de teste, diagrama de atividade textual, deve ser salvo.

3.1 Análise

Com a necessidade constante de garantia de qualidade do *software*, é essencial o uso de testes automatizados, por agilizarem e facilitarem o processo. Para suprir de uma maneira esse problema e os erros que são causados com a criação manual dos testes, foi desenvolvido um sistema capaz de criar os testes partindo da especificação dos requisitos.

No diagrama de caso de uso do sistema, são apresentadas as ações possíveis do usuário, testador do sistema. Para essas ações, foram definidas algumas verificações, como, por exemplo, quando o testador selecionar um arquivo é, então, realizada uma verificação garantindo que esse é um diagrama válido. O usuário ainda pode escolher onde deseja salvar o diagrama a ser gerado, que também passa por uma verificação para validar o diretório escolhido, e realizar a transformação do modelo indicado.

O testador, ator principal do diagrama, pode escolher a opção para gerar casos de teste, que executará a transformação entre modelos, gerando o diagrama de atividade com os casos de teste. Quando essa for selecionada, o arquivo indicado é, então, utilizado pelas classes em ATL, responsáveis pela transformação, para a geração do novo modelo, o diagrama com casos de teste.

Outro diagrama criado na análise do sistema pode ser visto na Figura 1, um diagrama de atividade representando o fluxo de ações possíveis a serem realizadas no *software* UML2UMLTesting. Primeiramente, o usuário deve selecionar um arquivo a ser transformado, caso essa ação seja cancelada, é retornado para essa ação, não sendo possível a

realização de outras sem antes essa estar concluída. Após, o usuário escolhe um local para salvar o arquivo, também não podendo executar a transformação sem antes o local de destino ter sido escolhido. Ao ser executada a transformação, o diagrama gerado é então salvo no local escolhido.

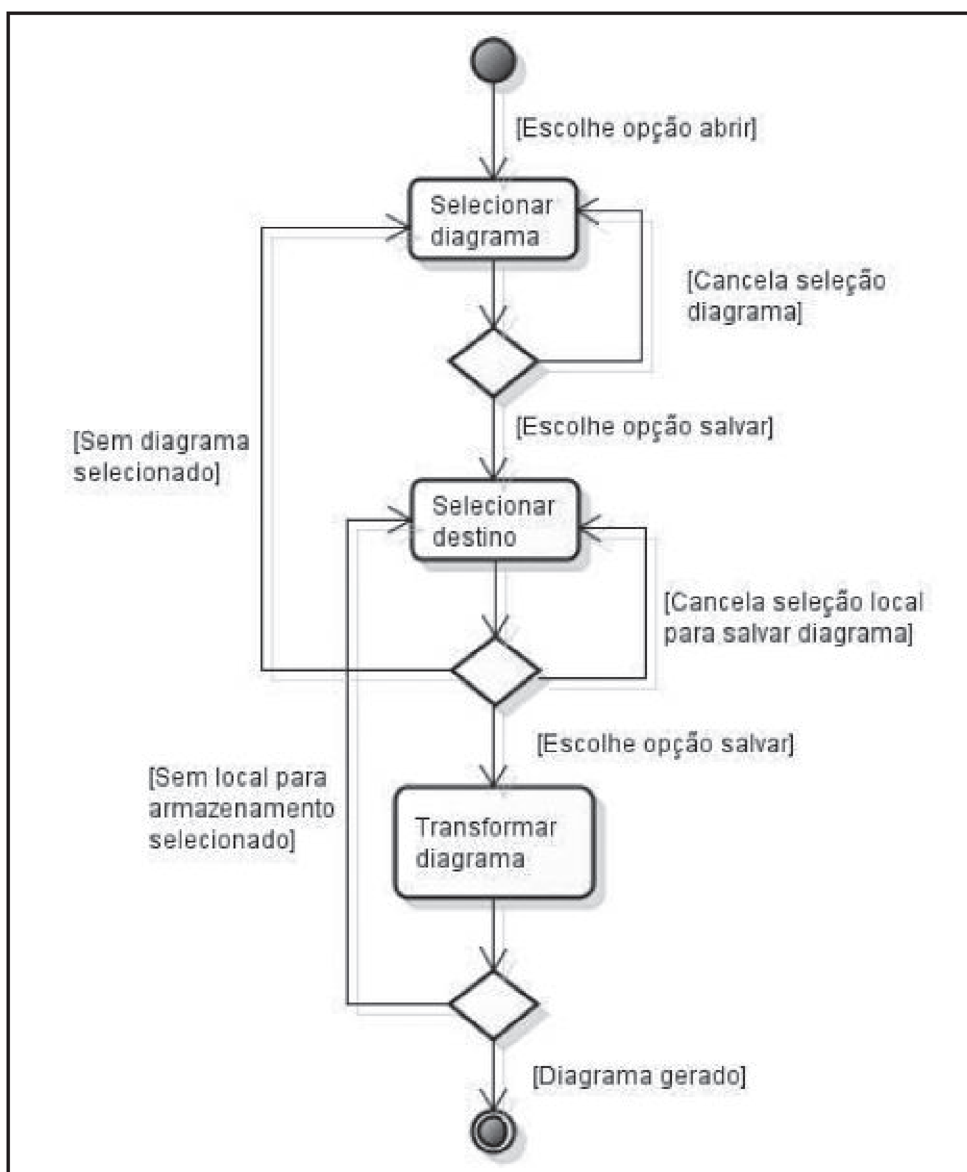

Figura 1 – Diagrama de atividade
Fonte: Elaborado pela autora (2012)

3.2 Desenvolvimento

Para o desenvolvimento do *software* UML2UMLTesting, foram utilizadas as linguagens de programação Java e de transformação entre modelos ATL. Ambas são compatíveis e suportadas pela ferramenta Eclipse, o qual foi utilizado na versão Juno juntamente com os *plugins* do projeto *Modeling* da plataforma Eclipse, que permite o desenvolvimento em ATL e ecore. Para o desenvolvimento da interface gráfica com Java, foi utilizada a biblioteca *Swing*.

Os metamodelos utilizados pelos códigos em ATL foram criados utilizando uma interface gráfica existente no Eclipse para a criação desses em Ecore. Para realizar as transformações entre modelos com o Java, foi necessário o uso do ATL *plugin*, um *plugin disponibilizado pela plataforma Eclipse*, que permite a criação de classes em Java que suportam chamadas para códigos em ATL e inicializam o *framework* EMF, necessário para as transformações.

Na Figura 2, é apresentado o fluxo em alto nível do comportamento do sistema criado, o UML2UMLTesting. Primeiramente, o usuário insere o diagrama de sequência, que é então repassado para os códigos ATL, que utilizam os metamodelos para criar o novo diagrama. Como resultado, é obtido o diagrama de atividade em formato textual, contendo os casos de teste, baseados no diagrama recebido.

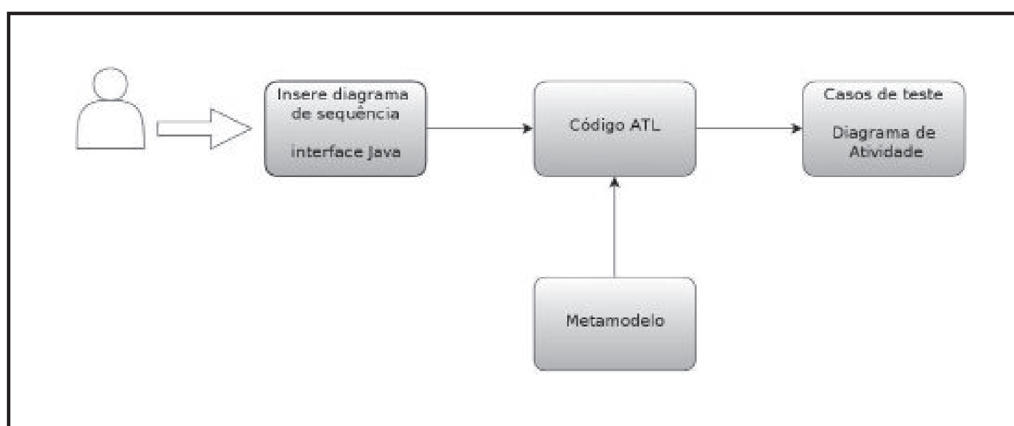

Figura 2 – Fluxo UML2UMLTesting
Fonte: Elaborado pela autora (2012)

3.2.1 Diagramas UML

Os diagramas utilizados no sistema UML2UMLTesting devem ser gerados a partir da ferramenta de modelagem Magic Draw, devido a essa dar suporte à exportação dos diagramas em UML2. O UML2UMLTesting recebe como entrada diagramas de sequência em formato UML, que são estruturados conforme o padrão XMI. Os modelos utilizados foram criados na ferramenta *Magic Draw* e exportados na mesma para arquivos em formato UML.

Dentro desses arquivos, há uma estrutura em XMI com todas as informações contidas no diagrama de sequência no formato gráfico, separadas por meio de *tags*[13], sendo assim possível a leitura desses a partir da linguagem de transformação entre modelos ATL. Esses modelos são lidos como sendo arquivos XMI pelo código, que extrai as informações necessárias conforme os metamodelos. Após os dados terem sido armazenados no metamodelo correspondente, é criado um novo arquivo com essas informações em formato UML contendo uma estrutura em XMI.

O diagrama de sequência que é recebido como modelo de entrada deve atender a algumas restrições que foram definidas para o correto funcionamento do sistema. Na especificação dos elementos *lifeline* do diagrama, os que correspondem ao ator e ao

[13] TAG: Marcação usada como palavra-chave para separar elementos.

sistema, apenas o sistema ou interface que será utilizada diretamente pelo testador ou usuário, devem ser nomeados como ator e SUT, respectivamente.

O sistema não possui suporte a diagramas de sequência criados a partir de outras ferramentas de modelagem diferentes do *Magic Draw*. Para que os casos de teste gerados tenham melhor cobertura do sistema, é importante que no modelo de entrada constem as mensagens do SUT para o ator quando no formato resposta.

Além disso, foram limitados os tipos de mensagens que podem ser utilizados nos diagramas de sequência utilizados no UML2UMLTesting. A ferramenta foi desenvolvida apenas para dar suporte a diagramas que possuem mensagens síncronas. Caso sejam utilizados diagramas com mensagens assíncronas, o resultado pode não corresponder ao esperado, gerando um modelo de saída não coerente.

O modelo final a ser gerado pela ferramenta consiste em um diagrama de atividade em representação textual, não sendo suportada a sua visualização gráfica e possuindo as mesmas *tags* que os modelos desse tipo gerados pela ferramenta *Magic Draw*, como *node* e *weight*.

3.2.2 Metamodelos

O *software* faz uso de cinco metamodelos no formato *ecore*. Esses metamodelos são usados por mais de uma classe ATL, para que os dados contidos em um possam ser passados para outros conforme o código ATL escrito.

Os metamodelos foram criados dentro da ferramenta Eclipse, utilizando os *plugins* do projeto *Modeling*, que já vêm de forma nativa em algumas versões desse *software*. Esses metamodelos são constituídos por classes, que podem ou não estarem relacionadas e podem possuir ou não atributos e operações. As classes do metamodelo funcionam como se fossem vetores. Ao adquirir informações para todos os seus atributos, essas informações são salvas em uma posição de memória e, ao ler novamente as informações do modelo de entrada, esses valores serão armazenados em uma nova posição desse "vetor". Para o sistema proposto, foram utilizadas classes sem operações e sem relacionamento entre elas.

No processo inicial, é utilizado o metamodelo *metamodelSequencia.ecore* (Figura 3), no qual são salvas informações adquiridas do modelo a ser transformado, diagrama de sequência, tendo como base outro metamodelo, um modelo disponibilizado pela ferramenta Magic Draw, que foi utilizada para a criação dos diagramas e que possui todas as informações sobre os elementos contidos no diagrama de sequência gerado por essa e lido pelo UML2UMLTesting.

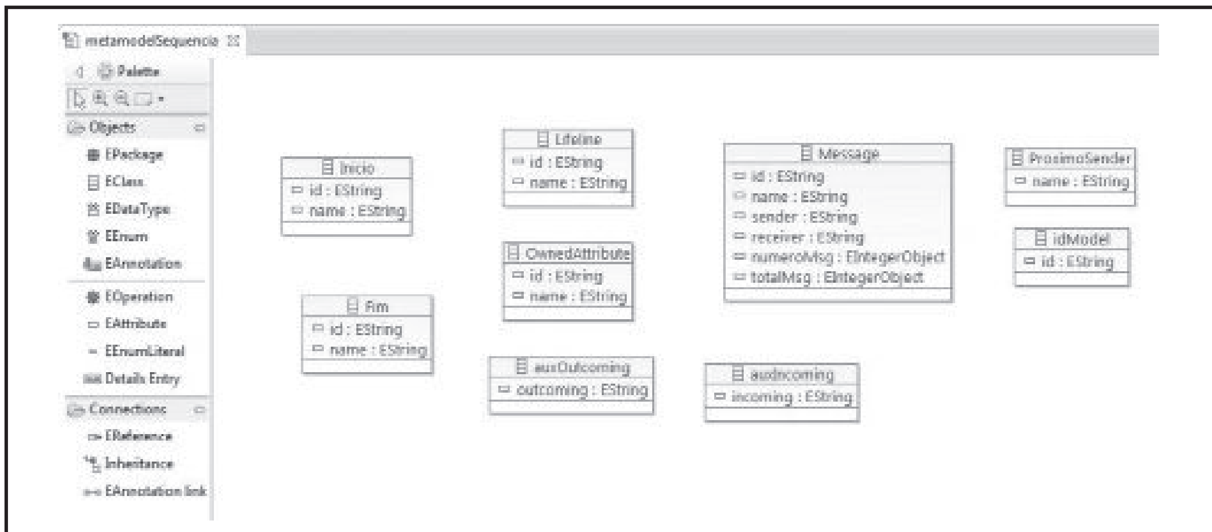

Figura 3 – Metamodelo metamodelSequencia.ecore
Fonte: Elaborado pela autora (2012).

O metamodelo *metamodelSequencia* irá extrair informações necessárias, conforme especificado no código desenvolvido em ATL, sobre os atores, conhecidas como *lifelines* em diagramas de sequência e sobre as mensagens enviadas e recebidas por esse. Para cada *lifeline* lido no modelo de entrada, o metamodelo armazenará somente os dados, *id*, nome, mensagens enviadas e recebidas, dos *lifeline* que estiverem nomeados como ator[14] e SUT[15], sendo os demais ignorados.

A classe *ownerAttribute* possui comportamento semelhante ao da classe lifeline, na qual são salvos valores para os atributos *id* e *name*, sendo o valor de *id* diferente para cada *tag* e o valor contido em *name* sendo igual ao do *lifeline* correspondente.

Na classe *Message*, são salvas as informações sobre as mensagens identificadas como pertencentes ao ator ou ao sistema, sendo os atributos *sender* e *receiver* utilizados para indicar quem enviou e quem recebeu a mensagem. Já os atributos *numeroMsg* e *totalMsg* são utilizados para armazenar o número atual da mensagem, que será utilizado no metamodelo seguinte para a criação das decisões e o total de mensagens encontradas respectivamente.

As classes *auxOutcoming* e *auxIncoming* são utilizadas para criar valores de *incoming* e *outcoming* para cada mensagem salva. Esses são atributos existentes em diagramas de atividade para indicar o fluxo das mensagens.

Já a classe *idModel* é utilizada para armazenar o *id* da *tag Model* do modelo de entrada. Esse valor será utilizado para criar os ids das classes *Início* e *Fim*, que serão utilizadas para criar as *tags* inicial e final do diagrama de atividade.

Com os dados necessários extraídos do modelo recebido, o metamodelo *metamodelSequencia* é então lido por outra classe em ATL, que será responsável por extrair e manipular os dados necessários conforme o metamodelo *decisions.ecore*.

O metamodelo decisions é utilizado apenas para a criação das decisões entre uma ação do usuário (ator) e do sistema (SUT). Essas decisões e a ordem dessas são

[14] Ator: O sistema foi projetado para aceitar como sendo o ator do diagrama, *lifelines* nomeados como: ator, ATOR, actor e ACTOR.
[15] SUT: Para o *lifeline* correspondente ao sistema são aceitos como nome: SUT, sut.

armazenadas no metamodelo decisions.

Após os dados necessários estarem conforme o metamodelo *decisions*, esse é então passado para outra classe que será responsável por extrair as informações e passar para o metamodelo *refDecisions.ecore*.

No metamodelo *refDecisions*, serão salvas somente as decisões necessárias, ou seja, as decisões que estiverem localizadas entre uma ação do ator e uma resposta ou ação do sistema. Esse metamodelo foi criado para remover possíveis decisões criadas entre ações do ator, sendo válidas somente decisões entre mensagens do ator e do SUT.

As informações do metamodelo *refDecisions* são então passadas para o metamodelo *messagesDecisions.ecore*. Esse *metamodelo*, ao contrário dos anteriores, receberá dados de dois *metamodelos,* do *decisions* e do *metamodelo metamodelSequencia*, em que as informações necessárias serão mescladas nesse novo *metamodelo*, que servirá de base para o metamodelo final.

O metamodelo *messagesDecisions*, é responsável por armazenar as mensagens e decisões criadas na ordem correta, ou seja, nesse serão armazenadas mensagens e decisões intercaladas conforme a ordem gerada no metamodelo anterior.

Por fim, é utilizado o metamodelo *metamodelAtividade.ecore*, que recebe as informações do metamodelo *messagesDecisions* e a partir dessas é montado o modelo de saída, um diagrama de atividade para casos de teste em formato textual.

3.2.3 Transformação entre modelos

O *software UML2UMLTesting* utiliza a linguagem de transformação de modelos ATL para executar a conversão do diagrama inserido pelo usuário para um novo diagrama contento os casos de teste. A escolha pela linguagem ATL se deve ao fato dessa ser compatível e pertencente à plataforma Eclipse, o que facilita o uso da mesma e a integração com a interface gráfica desenvolvida em Java.

Conforme Paes (2009), a ATL faz uso de *helpers* e *rules*. Os *helpers* funcionam como se fossem métodos em Java, em que podem ser alocados códigos que venham a ser usados em vários pontos do código. As *rules* por sua vez, são responsáveis pela execução da transformação, ou seja, é onde são definidos o tratamento e o destino dos dados lidos no modelo de entrada.

Na Figura 4, é apresentado um exemplo de código ATL. O *helper* mostrado na imagem é utilizado para a geração de um *id* que será utilizado para a identificação dos nodos inicial e final, existentes em diagramas de atividade. Logo após, é apresentada uma *entrypoint rule*, que consiste em uma regra que é executada somente uma vez, no início da execução do código. A *rule* mensagens é, então, responsável por obter somente as mensagens do ator e do SUT, conforme condições especificadas.

```
helper context UML!Model def: idNodo (d: String, e: String) : String =
    if (e = 'inicio') then
        d.concat ('_inicial')
    else
        d.concat ('_final')
    endif;

entrypoint rule incomingNodoInicio () {
    to
        w : metamodelSequencia!auxIncoming
        do {
            w.incoming <- UML!Model->allInstances().asSequence().first().idNodo
            (UML!Model->allInstances().asSequence().first().__xmiID__, 'inicio');
        }
}

rule mensagens {
    from
        a: UML!Message (
        if ((a.sendEvent.covered.asSequence().first().name = 'Actor' or
                a.sendEvent.covered.asSequence().first().name = 'ACTOR' or
                a.sendEvent.covered.asSequence().first().name = 'Ator' or
                a.sendEvent.covered.asSequence().first().name = 'ATOR') and
                (a.receiveEvent.covered.asSequence().first().name = 'SUT' or
                a.receiveEvent.covered.asSequence().first().name = 'sut')) then
                true
        else
```

Figura 4 – Exemplo código ATL
Fonte: Elaborado pela autora (2012).

Para a transformação do diagrama de sequência, obtido da especificação do sistema, para o diagrama de atividade contendo os casos de teste, foi necessário o uso de cinco classes ATL, sendo a primeira responsável por obter os dados do diagrama proposto e, conforme o metamodelo gerado pelo *Magic Draw* para a leitura de arquivos exportados pelo mesmo, armazená-los em um novo arquivo, também com extensão UML, de acordo com o metamodelo *metamodelSequencia.ecore*. Cada arquivo gerado corresponde a uma transformação, portanto, para chegar até a transformação alvo, foram necessárias quatro transformações intermediárias.

Durante a primeira transformação, foram extraídas do modelo de entrada apenas as informações que serão utilizadas para a criação do novo diagrama, como os nomes dos atores, seus *ids* para identificação e as mensagens trocadas entre os mesmos. Nesse primeiro código ATL, é feita a filtragem das mensagens, onde para cada mensagem lida é verificado se essa foi enviada pelo ator e recebida pelo SUT e vice-versa. Caso a mensagem não se enquadre nessa regra, ela é descartada e assim é lida a próxima.

Como em testes funcionais não é necessário saber quais iterações com outros sistemas ou métodos foram utilizados pelo sistema em testes, as mensagens trocadas entre o SUT e demais componentes do diagrama de sequência diferentes do Ator são ignoradas.

Como a proposta do sistema era gerar casos de teste, é necessário que haja verificações entre as ações do ator e do sistema para que possa ser garantido que o teste passou ou não. Para resolver isso, foi adotado o uso de decisões após as ações do ator

que tivessem como alvo o sistema. Assim, é possível, no teste, determinar se a resposta do SUT para determinada mensagem do ator está conforme o esperado, apresentado no diagrama gerado, indicando que o teste passou, ou caso não seja válida a resposta do sistema, indicando então que o teste falhou.

Na segunda transformação, a classe utilizada é responsável por analisar as mensagens salvas no arquivo gerado pela transformação anterior e acrescentar as decisões após as mensagens enviadas pelo ator. Porém, como em ATL não é possível ler dados que aparecem somente em algumas linhas do arquivo dentro de uma mesma *rule*, foi necessário o uso de uma nova classe, iniciando a terceira transformação intermediária, capaz de corrigir esse arquivo, removendo as decisões criadas que não seriam utilizadas.

A partir desse novo arquivo gerado, possuindo somente as decisões corretas, foi criada a quarta transformação, na qual, a partir de uma classe diferente, foram mescladas as mensagens enviadas e recebidas pelo ator e pelo SUT com as decisões criadas anteriormente. Nesse momento, fez-se uso de dois metamodelos e dois modelos como entrada, os arquivos gerados na primeira e na última transformação.

A última transformação consiste em gerar o diagrama de atividade com os casos de teste, cujos dados obtidos da transformação anterior são organizados conforme o metamodelo *metamodelAtividade.ecore*.

A interface gráfica proposta para o UML2UMLTesting foi feita utilizando a linguagem de programação Java, pois, além de ser compatível com a ferramenta de desenvolvimento Eclipse, é compatível com classes em ATL. O UML2UMLTesting possui uma barra de menu para navegação, possuindo como opções “abrir”, “salvar”, “ajuda” e “executar” a transformação entre modelos.

4 Experimentos

Para o início da implementação do *software UML2UMLTesting*, foram utilizados como base alguns exemplos de código ATL disponíveis no site Eclipse (2012) da ferramenta Eclipse. Além disso, durante o desenvolvimento do trabalho proposto, foram realizados testes a fim de verificar o andamento e o correto funcionamento.

Para os testes, foram criados alguns diagramas de sequência variados para garantir que apenas os dados necessários estavam sendo coletados. Nas seções seguintes, é apresentado um exemplo de diagrama utilizado para testes e os resultados obtidos durante o desenvolvimento e os testes.

4.1 Exemplos utilizados

Como o objetivo do *software* UML2UMLTesting é gerar casos de teste em formato de diagrama de atividade textual tendo como entrada diagramas de sequência em formato UML, sendo somente aceitas como válidas as mensagens trocadas entre o ator e o SUT do diagrama, visto que, para testes funcionais é necessário apenas considerar as funcionalidades do sistema, foram, então, necessários alguns testes com diferentes

formas, como diagramas com mais de um ator ou SUT, com mensagens enviadas para outros *lifelines* que não fossem aceitos e até mesmo mensagens de autodelegação.

Para o sistema, foi determinado que os nomes válidos para os *lifelines* do ator fossem escritos como ator, actor, ATOR e ACTOR, e para o SUT fossem aceitas as formas sut e SUT. Qualquer forma de escrita diferente dessas, assim como a inexistência de um desses, será ignorada pelo sistema, e esse não será aceito como um diagrama válido.

Foram, então, realizados testes com diagramas possuindo diferentes formas de escrita para o nome do ator e do SUT, assim como foram feitos testes utilizando mais de um ator e/ou SUT no mesmo diagrama. Porém, a ferramenta utilizada para a geração dos diagramas, Magic Draw, não permite que haja mais de um *lifeline* com o mesmo nome. Para tanto, foram utilizados os mesmos nomes, porém acrescidos de um número, o que fez com que o sistema ignorasse esses *lifelines* conforme o esperado.

Em uma primeira versão do sistema, durante um dos testes realizados, foi possível perceber que o sistema não estava conseguindo tratar corretamente o diagrama quando esse possuía duas mensagens seguidas do ator, ou seja, sem resposta por parte do SUT, gerando erroneamente decisões entre essas mensagens do ator, sendo o correto gerar decisões apenas entre as mensagens do ator para o SUT, a fim de testar se o comportamento esperado ocorreu.

Na Figura 5, é apresentado um exemplo de diagrama de sequência utilizado para a realização de testes. Nessa é apresentado um exemplo parecido com diagramas que podem vir a serem utilizados pelo sistema para a criação dos casos de teste.

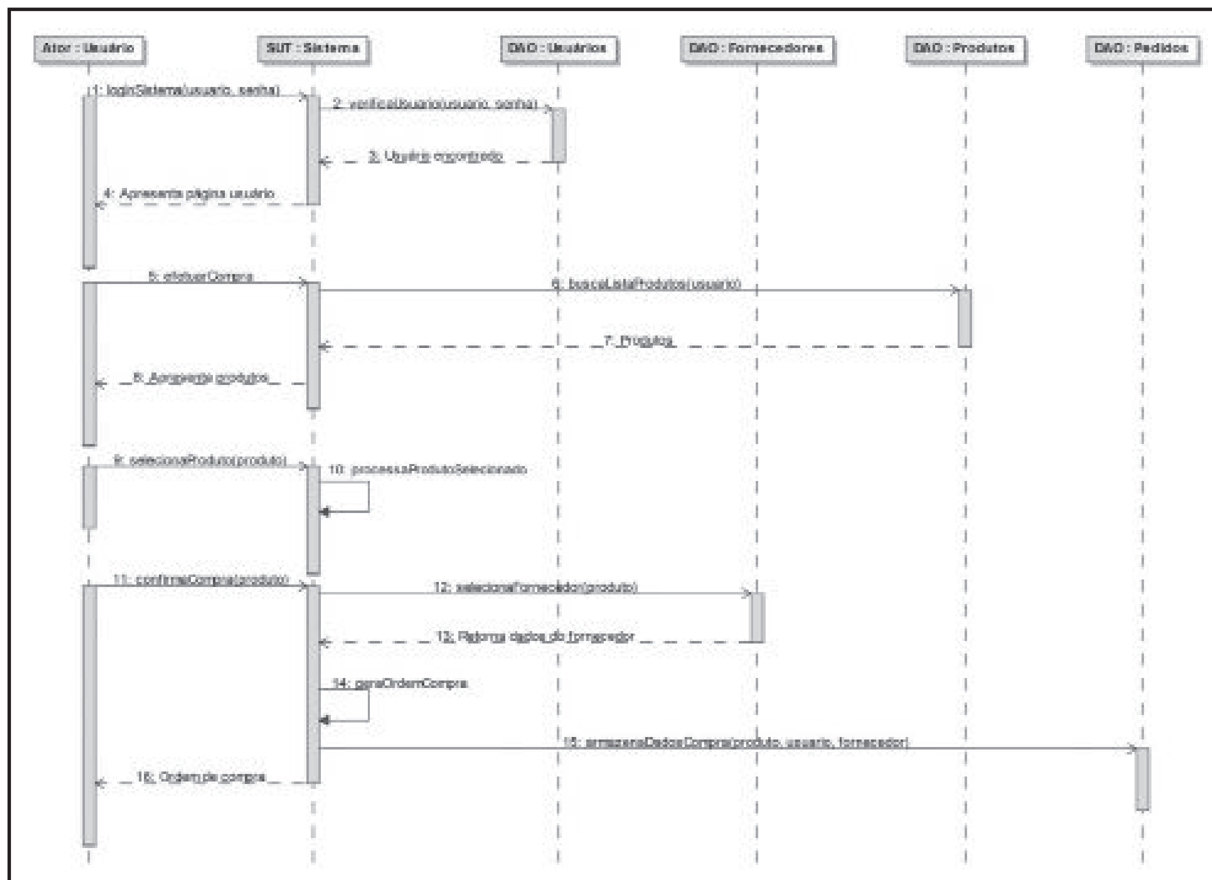

Figura 5 – Diagrama de sequência usado para testes
Fonte: Elaborado pela autora (2012).

Conforme apresentado no diagrama da Figura 5, o sistema recebe ações do ator, consideradas como mensagens e, a partir das solicitações feitas, são enviadas informações para outros *lifelines*, denominados de DAO e que representam tabelas em um banco de dados. O sistema, nomeado como SUT, envia respostas para o ator somente em alguns casos, com o intuito de garantir que as decisões para o caso de teste estão sendo colocadas somente entre ações do ator e o sistema.

A Figura 6 foi gerada também por meio da ferramenta Magic Draw a fim de representar graficamente os casos de teste gerados em formato textual pelo *software* UML2UMLTesting para o diagrama de sequência apresentado anteriormente na Figura 5. Esses casos de teste gerados textualmente podem ser utilizados para executar os testes ou para escrita de testes manuais em ferramentas de documentação ou de automatização de testes.

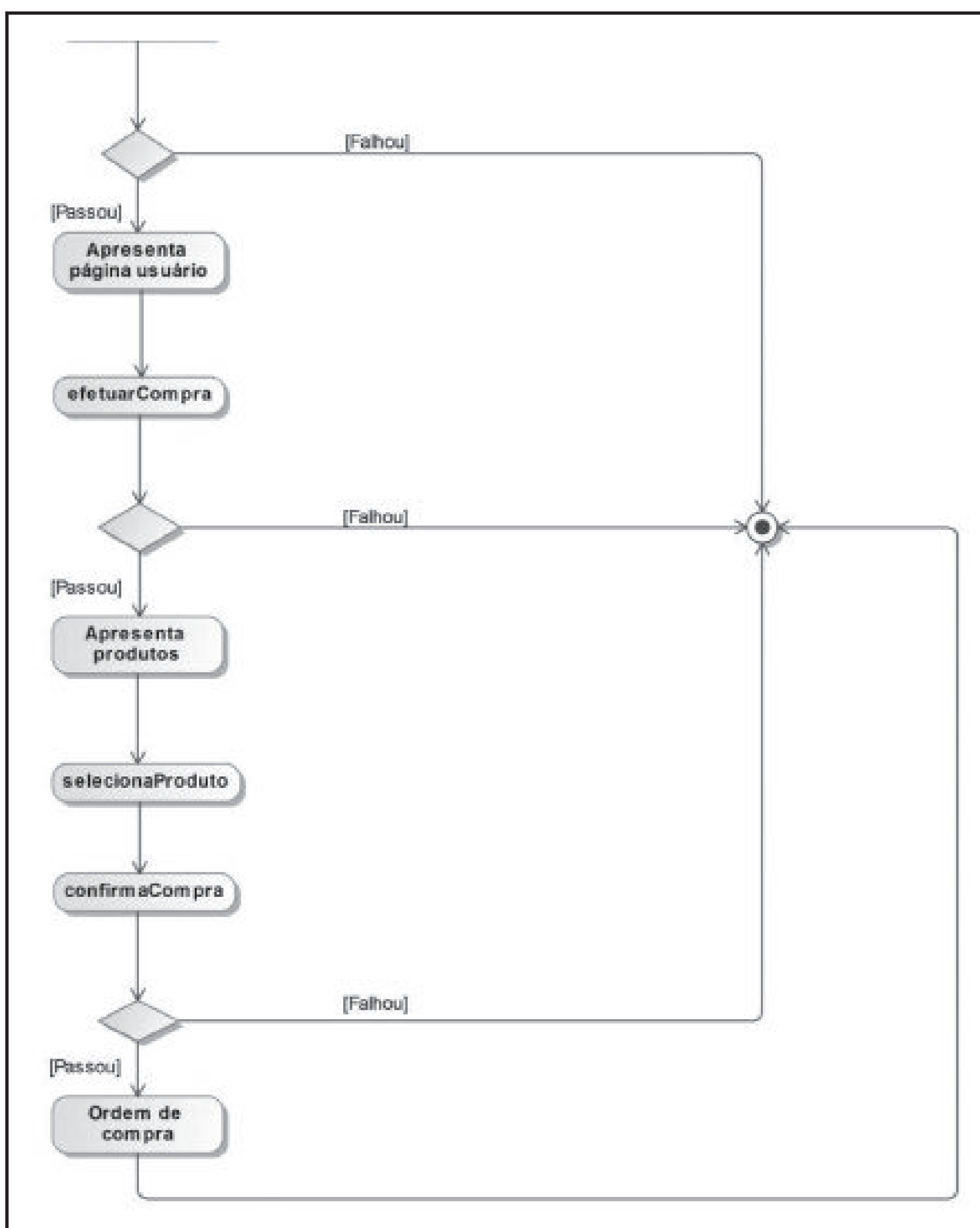

Figura 6 – Diagrama de atividade para diagrama de sequência anterior
Fonte: Elaborado pela autora (2012).

A partir da representação gráfica do diagrama de sequência anterior, é possível visualizar os casos de teste gerados. Como exemplo disso, após o usuário efetuar o *login* no sistema, o esperado é que o sistema apresente a página pertencente à conta desse usuário. Caso isso não ocorra, o teste é considerado como falho, ou seja, o status recebe o valor falhou, e conforme o diagrama da Figura 6, o teste é encerrado, sendo encaminhado para o estado final do diagrama de atividade.

Conforme descrito anteriormente, as decisões só devem ser utilizadas entre ações do ator e SUT. Sendo assim, após o usuário selecionar o produto para a compra, atividade “selecionaProduto”, o mesmo deve confirmar a compra, atividade “confirmaCompra”, sem que haja uma decisão entre essas duas atividades, já que ambas são realizadas pelo ator.

4.2 Resultados obtidos

Como resultado da pesquisa feita sobre os testes baseados em modelos, foi desenvolvido o *software* UML2UMLTesting, para gerar os casos de teste em formato de diagrama de atividade textual tendo como entrada diagramas de sequência criados na especificação do sistema que se pretende testar. A partir da pesquisa feita sobre o assunto, foi necessário o aprendizado sobre outras técnicas e também sobre a linguagem de transformação entre modelos, ATL.

Na Figura 7, é apresentado um exemplo de diagrama de atividade textual contendo os casos de teste gerado a partir do diagrama de sequência da Figura 5. Cada nodo representa uma ação, correspondente ao diagrama de sequência utilizado como entrada e, por meio das *tags edge*, é definido o fluxo dessa ação.


```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="20110701" xmlns:xmi="http://www.omg.org/spec/XMI/20110701" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xmlns="http://www.eclipse.org/uml2/4.0.0/UML" xsi:schemaLocation=
"http://www.eclipse.org/uml2/4.0.0/UML ../metamodelo/metamodelAtividade.ecore">
  <node xmi:id="_pqpvUDYqEeKFwL_xmbUBbA" xmi:type="uml:InitialNode" id="eee_1045467100313_135436_1_inicial" name=
  "inicio" visibility="public" outcoming="_15_5_dc40320_1348327814011_987143_447"/>
  <edge xmi:id="_pqpvUTYqEeKFwL_xmbUBbA" xmi:type="uml:ControlFlow" id="_15_5_dc40320_1348327814011_987143_447" name=
  "" visibility="public" source="eee_1045467100313_135436_1_inicial" target="_15_5_dc40320_1348327814011_987143_447"/>
  <weight xmi:id="_pqpvUjYqEeKFwL_xmbUBbA" xmi:type="uml:LiteralInteger" name="" value="1"/>
  <node xmi:id="_pqpvUzYqEeKFwL_xmbUBbA" xmi:type="uml:CallBehaviorAction" id=
  "_15_5_dc40320_1348327814011_987143_447" name="loginSistema" visibility="public" incoming=
  "eee_1045467100313_135436_1_inicial" outcoming="eee_1045467100313_135436_11"/>
  <edge xmi:id="_pqpvVDYqEeKFwL_xmbUBbA" xmi:type="uml:ControlFlow" id="eee_1045467100313_135436_11" name=""
  visibility="public" source="_15_5_dc40320_1348327814011_987143_447" target="eee_1045467100313_135436_11"/>
  <weight xmi:id="_pqpvVTYqEeKFwL_xmbUBbA" xmi:type="uml:LiteralInteger" name="" value="1"/>
  <node xmi:id="_pqpvVjYqEeKFwL_xmbUBbA" xmi:type="uml:DecisionNode" id="eee_1045467100313_135436_11" name="decisao"
  visibility="public" incoming="_15_5_dc40320_1348327814011_987143_447" outcoming=
  "_15_5_dc40320_1348327948815_476957_537 eee_1045467100313_135436_1_final"/>
  <edge xmi:id="_pqpvVzYqEeKFwL_xmbUBbA" xmi:type="uml:ControlFlow" id="_15_5_dc40320_1348327948815_476957_537" name=
  "passou" visibility="public" source="eee_1045467100313_135436_11" target="_15_5_dc40320_1348327948815_476957_537"/>
  <weight xmi:id="_pqpvWDYqEeKFwL_xmbUBbA" xmi:type="uml:LiteralInteger" name="" value="1"/>
  <node xmi:id="_pqpvWTYqEeKFwL_xmbUBbA" xmi:type="uml:CallBehaviorAction" id=
  "_15_5_dc40320_1348327948815_476957_537" name="Apresenta página usuário" visibility="public" incoming=
  "eee_1045467100313_135436_11" outcoming="_15_5_dc40320_1348327990824_271496_557"/>
```


Figura 7 – Diagrama de atividade textual gerado pelo UML2UMLTesting
Fonte: Elaborado pela autora (2012).

A partir dos diagramas utilizados para os testes, foi possível garantir que o objetivo do *software* proposto foi alcançado, já que esse está gerando os casos de teste por meio de um diagrama de atividade, no qual são usadas as mensagens trocadas entre o ator e o SUT, sendo as demais desconsideradas. Em cada situação que indique se o teste passou ou não, foi utilizado o elemento *decision*, ponto de verificação.

Durante o desenvolvimento, foi definido que cada teste que compõe o caso de teste possuiria uma verificação, indicada pelo elemento *decision* no diagrama, sempre após uma ação do ator que resultasse em uma ação do SUT. É possível citar como exemplo uma tela de *login*, na qual, em um diagrama de sequência, o ator (usuário) acessaria a página de *login* e inseriria seus dados, usuário e senha corretamente, e a resposta do SUT consistiria em apresentar a página correspondente à conta do usuário logado. Caso algum problema acontecesse, como, por exemplo, uma falha existente no sistema, a página do usuário não seria mostrada, indicando, assim, que um problema no sistema foi encontrado.

Com isso, sendo a conta do usuário apresentada, o resultado para o teste seria "passou". No caso de não ser apresentada a página da conta do usuário, o teste seria considerado como "falhou", pois o esperado não ocorreu. Tendo como base isso, o diagrama utiliza as decisões para indicar esses status.

Levando em conta o exemplo anterior, a decisão indicaria esse status, onde após uma ação do usuário, se essa obtiver a resposta esperada do SUT, o teste passou e o próximo passo é indicado, podendo esse ser considerado como um novo teste, em que haverá uma nova verificação do status. Caso o esperado não seja o mesmo que o ocorrido, o teste falhou, sendo o fluxo apontado para o final do diagrama, indicando assim que o teste não possui mais continuidade. Esse caso também pode ser utilizado para testes que dependem ou que só podem ser executados se o teste anterior obtiver sucesso na execução.

Na Figura 7, é apresentado graficamente um exemplo do diagrama de atividade textual que é gerado pelo UML2UML2Testing. Por meio dessa, é possível ver uma representação clara do comportamento do teste e da utilidade das decisões para os testes.

Ainda conforme a Figura 7, primeiramente, é executada a ação do ator, "loginSistema", que tem como sequência de fluxo uma decisão, onde é verificado se após a execução da *login* no sistema o obtido do SUT foi a apresentação da página do usuário, indicando assim que o teste passou, ou se o obtido não corresponde ao esperado, "Apresenta página usuário", indicando que o teste falhou. Esse é então encerrado, devendo ser repassado para os desenvolvedores o erro encontrado para que esse possa ser corrigido.

Com a página do usuário já aberta, esse pode seguir para o teste seguinte que consiste na atividade "efetuarCompra". Após a atividade de efetuar a compra ser realizada, é executada uma nova verificação para garantir o resultado do teste. Caso os produtos existentes e disponíveis para esse usuário sejam apresentados, ação do SUT "Apresenta produtos", o teste é considerado como tendo passado e, então, o próximo teste deve ser realizado. No caso de o esperado não ser obtido, o teste é considerado com o status de falhou e sendo então finalizado.

5 Conclusão

No presente trabalho, foi descrito o desenvolvimento do *software* UML2UML-Testing para a geração de casos de teste de forma automatizada, utilizando diagramas de sequência em UML como entrada. Nesse é possível indicar um dos diagramas de sequência criados durante a especificação do sistema e, então, gerar casos de teste para que possa ser possível testar as funcionalidades descritas no modelo disponibilizado. O diagrama de atividade gerado como saída não possui representação gráfica, apenas formato textual, consistindo em um arquivo com extensão UML, que pode ser lido em qualquer editor de texto.

Os casos de teste gerados podem ser utilizados pelo testador de *software* para executar os testes do sistema, como pode também utilizá-los como base para a criação dos testes automatizados em ferramentas específicas para esse fim. Por ser uma ferramenta *desktop* desenvolvida em Java, o UML2UMLTesting pode ser executado em diferentes plataformas operacionais, além de não depender da viabilidade de conexão com a internet, o que em alguns casos poderia dificultar o uso da ferramenta.

O UML2UMLTesting tem como objetivo facilitar a criação dos testes, evitar erros causados durante esse processo, aumentar a qualidade e reduzir o tempo gasto com essa atividade, já que os casos de teste são criados utilizando os diagramas da especificação do sistema e não somente após o término do ciclo de desenvolvimento. Além disso, como a ferramenta permite a criação dos testes baseados nos modelos da especificação do sistema em testes, as chances de haver erros decorrentes de uma interpretação incorreta do sistema por parte do testador são reduzidas.

Como o UML2UMLTesting permite que os casos de teste sejam criados a partir de diagramas de sequência gerados durante a especificação do sistema, pode-se considerar que foram utilizados conceitos de testes baseados em especificação, juntamente com os conceitos de MDA, para o desenvolvimento do *software*, sendo os conceitos de MBT utilizados como base de pesquisa.

Com os testes sendo criados baseados na especificação, é possível garantir que o sistema desenvolvido está correspondendo ao esperado, além de evitar que os testes sejam criados somente após o sistema estar completo. Como, em alguns casos, os testes são criados ao final do desenvolvimento, acabam sendo criados testes tendenciosos, já que, para a criação desses, o testador pode vir a questionar o funcionamento correto do sistema para quem o desenvolveu, e no caso de o desenvolvedor ter interpretado a especificação de uma forma incorreta, os testes, mesmo obtendo sucesso durante a execução, estarão errados. A partir disso, a importância de criação de testes baseados na especificação fica evidenciada.

A implementação atual da ferramenta UML2UMLTesting constitui em um protótipo que permite o desenvolvimento de algumas melhorias como trabalho futuro a ser realizado, como, por exemplo, a geração de diagramas em formato gráfico e não apenas textual.

Referências

ALHIR, S. S. ***Guide to Applying the UML***. Nova Iorque: Springer, 2002.

BURNSTEIN, I. ***Practical Software Testing***: *A Process-Oriented Approach*. Nova Iorque: Springer, 2002.

CARTAXO, E. G. **Geração de Casos de Teste Funcional para Aplicações de Celulares**. 2006. 146 f. Dissertação (Mestrado em Ciências da Computação) - Programa de Pós-Graduação em Ciências da Computação, Universidade Federal de Campina Grande, Campina Grande, 2006.

ECLIPSE. **ATL** - *a model transformation technology*. Disponível em: <http://www.eclipse.org/atl/>. Acesso em: 28 ago. 2012.

FRANKEL, D. S. ***Model Driven Architecture***: *Applying MDA to Enterprise Computing*. Indianapolis: Wiley Publishing, 2003.

GROSE, T. J.; DONEY, G. C.; BRODSKY, S. A. ***Mastering XML***: *Java Programming With XMI, XML and UML.* [S.l]: John Wiley & Sons, 2002.

INRIA ATLAS. ***ATL Inventory***. 2006. Disponível em: <http://www.eclipse.org/m2m/atl/doc/ATL_Inventory.pdf>. Acesso em: 28 ago. 2012.

JACKY, J. *et al.* ***Model-Based Software Testing and Analysis with C#***. Nova Iorque: Cambridge University Press, 2008.

KLEPPE, A.; WARMER, J.; BAST, W. ***MDA Explained***: *The Model Driven Architecture - Practice and Promise*. Boston: Addison-Wesley, 2003.

LAMANCHA, B. P. *et al.* Automated Model-based Testing using the UML Testing Profile and QVT. ***Sixth MoDeVVa workshop associated with MODELS'09***. *Model-Driven Engineering, Verification and Validation: Integrating Verification and Validation in MDE*. Denver, 2009.

LARMAN, C. ***Applying UML and Patterns***: *An Introduction to Object-Oriented Analysis and Design and the Unified Process*. 2. ed. Prentice Hall PTR, 2001.

LIMA, A. S. **UML 2.3**: Do Requisito à Solução. São Paulo: Érica, 2011.

LINDLAR, F.; WINDISCH, A.; WEGENER, J. *Integrating Model-Based Testing with Evolutionary Functional Testing*. ***Third International Conference on Software Testing, Verification, and Validation Workshops***, 2010.

MELLOR, S. J. *et al.* **MDA Destilada**: Princípios de Arquitetura Orientada por Modelos. Rio de Janeiro: Ciência Moderna, 2005.

OMG. ***MDA Guide Version 1.0.1***. 2003. Disponível em: <http://www.omg.org/cgi-bin/doc?omg/03-06-01>. Acesso em: 24 ago. 2012.

PAES, J. *Model Transformation com ATL:* Pondo em Prática a M2M *Model Transformation*. **Revista Engenharia de *Software***, v. 1, n. 9, p. 36-44, 2009.

PENDER, T. A. ***UML Weekend Crash Course***. Indianapolis: Wiley Publishing, 2002.

PERRY, W. E. ***Effective Methods for Software Testing***. 3. ed. Indianapolis: Wiley Publishing, 2006.

PRESSMAN, R. S. **Engenharia de *Software***. 6. ed. Rio de Janeiro: McGraw-Hill, 2006.

RECH, J.; BUNSE, C. ***Model-Driven Software Development***: *Integrating Quality Assurance*. Nova Iorque: Information Science Reference, 2009.

REZA, H.; LANDE, S. *Model Based Testing Using Software Architecture*. ***Seventh International Conference on Information Technology*,** 2010.

RIOS, E.; MOREIRA FILHO, T. **Teste de *Software***. 2. ed. Rio de Janeiro: Alta Books, 2006.

SOUZA, I. F. C.; ARAÚJO, M. A. P. MDA - Arquitetura Orientada por Modelos: Um Exemplo Prático. **Revista Engenharia de *Software***, v. 1, n. 9, p. 28-34, 2009.