Automação da Técnica de Inspeção Guiada Usando MDA e Simulação de Modelos

Anne C. O. Rocha[1], Patrícia D. L. Machado[1], Franklin Ramalho[1]

[1]Grupo de Métodos Formais/UFCG, Campina Grande, PB – Brasil

{annec,patricia,franklin}@dsc.ufcg.edu.br

Abstract. *In a software development process, artifacts from a stage are used as input to create new artifacts on another. The transition between different artifacts may not be precise; inconsistencies may occur. These inconsistent artifacts may produce software with defects. In this context, a software inspection technique is needed to validate these artifacts. This paper presents a technique to automate a guided inspection technique, which evaluates the conformity between artifacts using test cases. As support for the automation, we are using MDA (Model Driven Architecture) to perform model-to-model transformations and the USE tool for model simulation.*

Resumo. *Em um processo de desenvolvimento de software, artefatos de uma etapa são utilizados como fonte para criação de novos artefatos. Então, defeitos podem ser inseridos durante a transição de uma etapa para outra: artefatos podem ficar inconsistentes, levando à construção de um software com defeitos. Neste contexto, uma técnica de inspeção é necessária para validar esses artefatos. Este artigo apresenta uma técnica de automação da técnica de inspeção guiada, a qual avalia a conformidade entre artefatos de diferentes níveis de abstração, utilizando casos de teste. Para dar suporte à automação, utilizamos conceitos de MDA (Model Driven Architecture) para transformação entre modelos e a ferramenta USE para simulação de modelos.*

1. Introdução

Um processo de desenvolvimento de software possui diversas etapas. Geralmente, um processo iterativo engloba as seguintes etapas: requisitos, projeto, codificação, validação, produção e evolução [Sommerville 2007]. Os artefatos produzidos por uma etapa geralmente são utilizados como fonte para as próximas etapas, podendo ocorrer erros nos pontos de transição entre as etapas do processo. Assim, a especificação de requisitos, os modelos de projeto e o código-fonte podem possuir inconsistências, defeitos, omissões ou anomalias. Para minimizar os defeitos encontrados nos artefatos de software durante o processo de desenvolvimento, existem as atividades de verificação e validação (V & V), por exemplo, testes e inspeção de software [Sommerville 2007].

Geralmente, há uma maior preocupação com os defeitos na etapa de codificação, onde existe alguma versão do software, possibilitando a realização de testes. Entretanto, os defeitos podem estar presentes desde as etapas iniciais do processo de desenvolvimento e não serem percebidos. Por isso, a técnica de inspeção de software é importante por analisar diferentes artefatos do processo de desenvolvimento sem

depender da execução do código [Kamperman 2003]. Quanto mais tarde defeitos forem encontrados em artefatos, maior será o custo para corrigi-los [Hutcheson 2003].

Existem várias técnicas de inspeção que avaliam a conformidade entre os artefatos de software de alto nível, como por exemplo, a técnica de Inspeção Guiada [McGregor e Sykes 2001], que é uma técnica que inspeciona manualmente diagramas UML (*Unified Modeling Language*) [OMG 2007] a partir de uma especificação base. A Inspeção Guiada é realizada com o auxílio de casos de teste, que são criados a partir de uma especificação com maior nível de abstração (p.e., uma especificação de requisitos) em relação ao nível de abstração dos artefatos que se quer inspecionar (p.e., diagramas de projeto). Estes casos de teste são utilizados para inspecionar os artefatos de forma a verificar a conformidade entre estes do ponto de vista do comportamento esperado [Major e McGregor 1999]. No entanto, realizar inspeção manual é um processo muito trabalhoso, podendo aumentar o custo do desenvolvimento, além de estar sujeito a erros.

Uma das alternativas ao processo de inspeção seria considerar a transformação automática de modelos em uma arquitetura MDA (*Model Driven Architecture*) que, teoricamente, eliminaria os problemas supracitados. No entanto, a geração completa de modelos de projeto a partir de documentos de requisitos não é sempre possível, visto que projeto é um processo decisório não totalmente passível de automação. Desta forma, a inspeção guiada é uma técnica importante até mesmo dentro do contexto de MDA.

Este artigo apresenta uma técnica que visa aprimorar a Inspeção Guiada de forma que ela possa ser realizada de maneira parcialmente automática, viabilizando a prática de inspeção dentro do processo de desenvolvimento. É dado enfoque à inspeção de diagramas UML com relação a documentos de requisitos. Para realizar esta automação foram utilizados conceitos como Semântica de Ações de UML2 [OMG 2007], além de técnicas de transformação de MDA (*Model Driven Architecture*) [Kleppe et al. 2003] e a ferramenta USE (*UML-based Specification Environment*) para simulação de modelos UML [Gogolla et al. 2007].

O artigo está estruturado da seguinte forma. A Seção 2 apresenta conceitos básicos sobre as técnicas e ferramentas utilizadas para realizar a nossa solução. A Seção 3 explica como é realizada a automação da Inspeção Guiada. Na Seção 4, um exemplo é apresentado para ilustrar a aplicação da técnica. A Seção 5 apresenta os trabalhos relacionados. Na Seção 6, as conclusões e trabalhos futuros são apresentados.

2. Fundamentação Teórica

Nesta seção, são apresentados alguns conceitos básicos necessários para uma melhor compreensão deste artigo, juntamente com o domínio do exemplo utilizado.

2.1. Inspeção Guiada

Técnicas de inspeção tradicionais geralmente são realizadas utilizando um documento que consiste de um conjunto de instruções e regras, que auxiliam o inspetor durante a inspeção[Kamperman 2003]. No entanto, estas regras não focam usualmente no comportamento do sistema. A técnica de Inspeção Guiada tem o diferencial de realizar a inspeção de forma a avaliar se os comportamentos presentes na especificação de requisitos estão presentes em todos os artefatos de forma consistente, completa e correta [Major e McGregor 1999].

Em testes de software, casos de teste são utilizados para exercitar o comportamento do sistema em diferentes cenários e testar se o sistema está de acordo com o comportamento esperado [Jorgensen 1995]. Desta forma, a técnica de Inspeção Guiada utiliza casos de teste para exercitar o comportamento dos diagramas UML. O processo para realizar a Inspeção Guiada manualmente pode ser visualizado na Figura 1 e é composto das seguintes etapas [Major e McGregor 1999]:

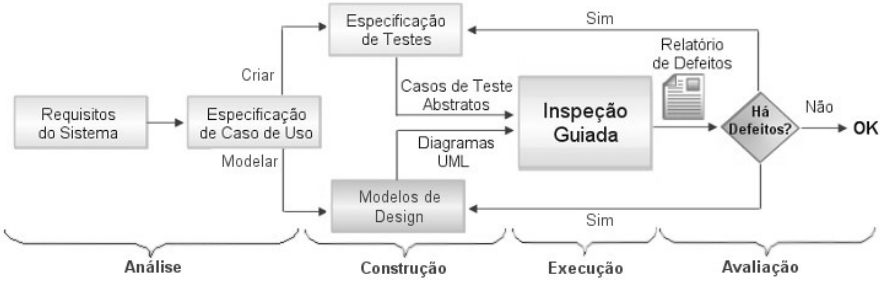

Figura 1: Processo de Inspeção Guiada.

1. *Análise*: o inspetor deve analisar a especificação de caso de uso para classificar cada caso de uso de acordo com seu grau de criticalidade para o sistema, que é proporcional à freqüência daquele caso de uso.
2. *Construção*: o inspetor deve construir uma especificação de testes, que é composta por um conjunto de casos de teste abstratos para cada caso de uso. Estes são usualmente descritos em linguagem natural quando derivados de requisitos.
3. *Execução*: a inspeção deve ser realizada de forma que os casos de teste sejam executados sobre cada diagrama UML, como, por exemplo, sobre um diagrama de seqüência, que represente o comportamento de um caso de uso.
4. *Avaliação*: o inspetor deve avaliar se, ao final da execução do Caso de Teste, o comportamento do caso de teste estava presente no diagrama inspecionado.

2.2. Semântica de Ações

Uma ação representa uma unidade fundamental do comportamento de uma parte da especificação. Ela pode converter um conjunto de entradas em um conjunto de saídas modificando um estado do sistema [OMG 2007]. Semântica de ações é um padrão, definido pela OMG, independente de plataforma, pois não possui uma linguagem concreta definida. Este padrão é um meta-modelo, que faz parte da especificação de UML2, o qual possui uma semântica bem definida com seus atributos, associações e restrições. Com isso, é possível descrever a semântica das ações de um sistema orientado a objetos, permitindo a geração automática de código em qualquer linguagem. As ações podem ser do tipo criação ou destruição de objetos, leitura ou escrita de variáveis, chamada de eventos ou operações, associação entre objetos, entre outras.

2.3. Simulação de Modelos

As ferramentas que realizam simulação em modelos UML têm a finalidade de transformar um modelo UML (diagrama de classe, diagrama de seqüência, etc) em um modelo executável. Além disso, estas ferramentas permitem que os modelos sejam validados durante sua execução através de restrições (invariantes, pré e pós-condições)

definidas em OCL (*Object Constraint Language*) [OMG 2005]. Na literatura existem algumas ferramentas para esta finalidade, como por exemplo, UMLAUT (*Unified Modeling Language All pUrposes Transformer*) [Ho et al. 1999], UMLAnt (*UML Animator and Tester*) [Trong et al. 2005] e USE (*UML-based Specification Environment*) [Gogolla et al. 2007]. Dentre estas ferramentas, em nossa técnica, a ferramenta USE foi utilizada para realizar a simulação dos modelos UML. Esta ferramenta foi escolhida por ser um software livre e por permitir a geração automática de diagramas de seqüência, que representam o comportamento do sistema. Caso alguma invariante não seja satisfeita, a ferramenta apresenta uma mensagem de alerta indicando o ponto do modelo que deve ser corrigido. Caso alguma pré/pós-condição das operações não seja satisfeita, a mensagem de alerta é exibida com uma linha de mensagem de cor vermelha no diagrama de seqüência gerado pelo USE durante a execução das operações.

2.4. Sistema de Jogo de Xadrez

O exemplo que foi utilizado para aplicar nossa técnica é referente a um sistema de jogo de xadrez. O xadrez é um jogo de tabuleiro, que possui regras para o movimento das peças e só pode ser jogado por 2 jogadores. O objetivo do jogo é analisar e imaginar estratégias para conseguir atacar o rei adversário de forma que seja um xeque-mate, ou seja, que o rei não possa se livrar do ataque. A Figura 2 apresenta o diagrama de classes que representa o sistema. A classe "JogoXadrez" inicia o jogo e permite cadastrar os jogadores em uma nova partida.

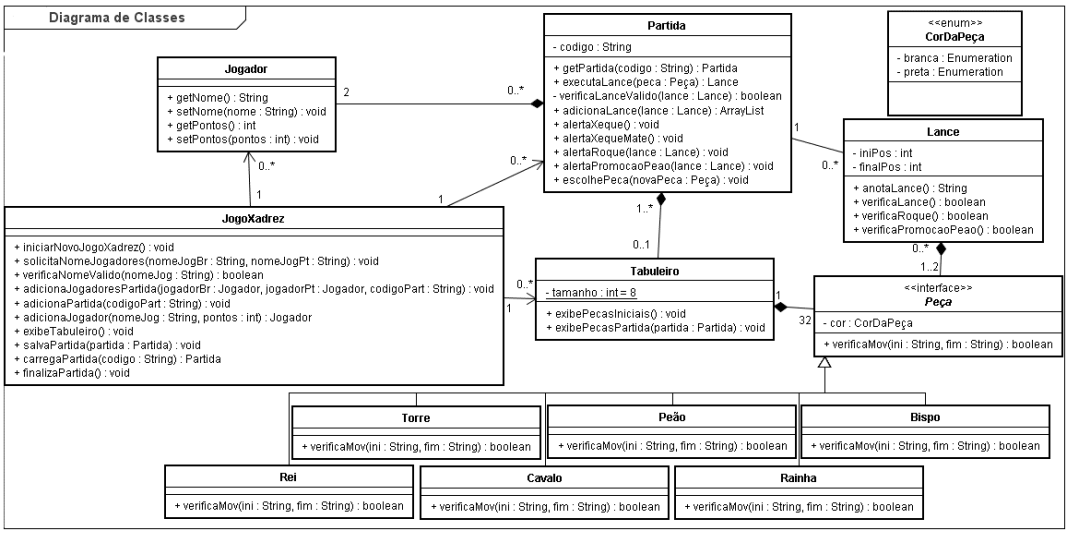

Figura 2: Diagrama de Classes para a aplicação de jogo de xadrez.

3. Automação da Técnica de Inspeção Guiada

Uma das grandes dificuldades no processo de inspeção guiada convencional é a "execução" do caso de teste em si. Para tal, é necessário fazer um mapeamento de cada passo do caso de teste em uma realização detalhada aceitável no diagrama de seqüência. Este é um processo difícil e bastante sujeito a erros, pois, usualmente, este mapeamento é feito apenas mentalmente. Desta forma, um dos desafios da automação é tratar a distância semântica entre os dois níveis abstratos a fim de tornar o processo mais

eficiente e efetivo. Outra dificuldade é que a técnica convencional não trata o problema da validação dos casos de teste em si. Isto pode afetar a confiabilidade dos resultados obtidos. A solução proposta a seguir trata estas dificuldades.

A Figura 3 apresenta uma visão geral da nossa técnica proposta para automação da Inspeção Guiada. Para realizar a Inspeção Guiada são necessários os seguintes documentos: a especificação de testes (casos de teste abstratos) e os modelos UML de projeto da aplicação (diagrama de classes e diagrama de seqüência). Estes documentos são criados a partir da especificação de caso de uso de forma manual. Assim, nossa técnica propõe os seguintes passos para automação da Inspeção Guiada: (1) anotar a semântica de cada passo dos casos de teste abstratos usando semântica de ações e o diagrama de classes; (2) utilizar técnicas de MDA para transformar um caso de teste anotado com semântica de ações em um caso de teste executável; (3) executar os casos de teste na ferramenta USE e gerar automaticamente diagramas de seqüência que representam os casos de teste abstratos; (4) realizar a inspeção no diagrama de seqüência de projeto utilizando como base o diagrama de seqüência do caso de teste gerado pelo USE. Ao final do passo 4, deve ser gerado um relatório de defeitos indicando os pontos de inconsistência no diagrama de seqüência inspecionado. Nas próximas subseções, é apresentado com detalhes como foi realizado cada passo da automação.

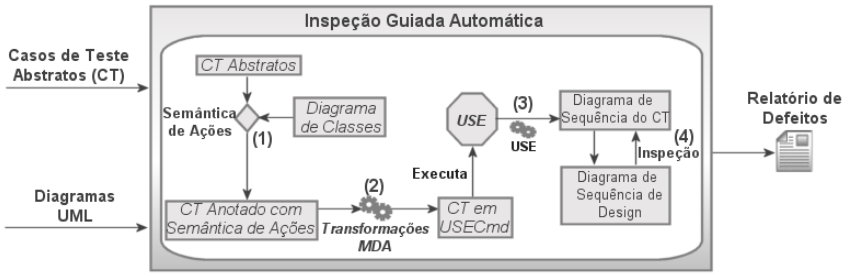

Figura 3: Passos para automação da Inspeção Guiada.

3.1. Passo 1: Anotando os Casos de teste com Semântica de Ações

De acordo com a técnica de Inspeção Guiada convencional, é necessário verificar se o comportamento de cada passo de um caso de teste está presente nos respectivos diagramas UML. Então, o primeiro passo para a automação da Inspeção Guiada foi identificar a semântica de cada passo de execução do sistema dos casos de teste abstratos, anotando-os com Semântica de Ações de UML2 [OMG 2007]. Essa anotação realizada nos casos de teste abstratos permite transformá-los em casos de teste executáveis, os quais podem ser descritos numa determinada linguagem concreta.

O procedimento utilizado para inserir a semântica de ações nos casos de teste é manual. Inicialmente selecionamos algumas ações da especificação de UML2 que poderiam ser necessárias para formalizar os casos de teste. Além disso, estendemos estas ações com informações que permitem identificar qual entidade do sistema será modificada por aquela ação. A Tabela 1 apresenta as ações que foram selecionadas com suas respectivas extensões. As informações complementares foram adicionadas com o auxílio do diagrama de classes da aplicação, que possui os nomes das classes e seus respectivos atributos e métodos.

Tabela 1: Semântica de ações selecionadas com extensão.

Semântica de Ações com Extensão	Descrição da Semântica
<CreateObjectAction> object | ObjectType	Ação de criar um objeto.
<DestroyObjectAction> object	Ação de remover um objeto.
<WriteVariableAction> object | variable | value	Ação de alterar o valor de uma variável.
<ReadVariableAction> object | variable	Ação de recuperar o valor de uma variável.
<CreateLinkObjectAction> object1 | link | object2	Ação de criar uma associação entre dois objetos.
<DestroyLinkAction> object1 | link | object2	Ação de remover a associação entre dois objetos.
<CallOperationAction> object | operation | parameters	Ação de chamada de execução de uma operação.
<ReplyAction> [return]	Ação de retorno de uma operação.

O resultado de um caso de teste anotado com semântica de ações é ilustrado no exemplo apresentado na Tabela 2. Por exemplo, o passo 1 do caso de teste abstrato está relacionado com as semânticas de ação "*CreateObjectAction*" e "*CallOperationAction*", pois é necessário criar um objeto da classe "JogoXadrez" para fazer a chamada da operação de iniciar o jogo. O passo 3 do caso de teste não foi anotado com uma ação semântica porque representa uma ação do usuário.

Tabela 2: Exemplo de um Caso de Teste com Semântica de Ações.

Caso de Uso: Iniciar jogo de xadrez **Dependências:** Caso de uso – Adicionar jogadores		**Freqüência:** Alta **Grau de Criticidade:** Alto	
#	**Passos do Caso de Teste Abstrato**	**Caso de Teste Anotado com Semântica de Ações**	**Resultado Esperado**
1	1. O sistema inicia um novo jogo de xadrez.	1. `<CreateObjectAction> jogo JogoXadrez` 1.a. `<CallOperationAction> jogo iniciarNovoJogoXadrez()`	Os jogadores são adicionados no jogo com sucesso.
	2. O sistema solicita os nomes dos 2 jogadores.	2. `<CallOperationAction> jogo solicitaNomeJogadores('Jonas','Maria')`	
	3. O usuário informa os nomes dos jogadores.		
	4. O sistema adiciona os 2 jogadores no jogo.	4. `<CallOperationAction> jogo adicionaJogador('Jonas', 0)` 4.a. `<CreateObjectAction> jogBr Jogador` 4.b. `<CallOperationAction> jogo adicionaJogador('Maria', 0)` 4.c. `<CreateObjectAction> jogPt Jogador`	

3.2. Passo 2: Usando transformações MDA para gerar casos de teste executáveis

Neste passo, a ferramenta USE é aplicada para simulação dos casos de teste. Para tal, transformamos automaticamente os casos de teste anotados com semântica de ações para a sintaxe dos comandos da ferramenta. Essa sintaxe possui algumas semelhanças com as ações selecionadas para representar os passos de execução dos casos de teste abstratos. A Tabela 3 apresenta um exemplo da sintaxe de alguns comandos de USE e a semântica de ações equivalente.

Tabela 3: Exemplo da relação entre a semântica de ações e os comandos USE.

Semântica de Ações	Sintaxe dos Comandos de USE
<CreateObjectAction> object | ObjectType	!create object : ObjectType
<WriteVariableAction> object | variable | value	!set object.attribute := value
<CreateLinkObjectAction> object1 | link | object2	!insert (object1, object2) into Association
<CallOperationAction> object | operation | parameters	!openter object operation()

A transformação é realizada utilizando o *framework* MDA (*Model Driven Architecture*) [Kleppe 2003], que permite transformar modelos abstratos em outros modelos através de regras de transformação sobre os meta-modelos. Estas regras foram descritas em ATL (*ATLAS Transformation Language*) [ATL 2008]. O procedimento

para transformação dos modelos é composto por 4 etapas: (i) criação dos meta-modelos de entrada e de saída; (ii) definição das regras de transformação em ATL; (iii) criação da instância do modelo de entrada; e (iv) geração do modelo de saída.

Na etapa de criação dos meta-modelos de entrada e saída, o meta-modelo de semântica de ações foi utilizado como entrada, que é o mesmo disponibilizado pelo OMG [OMG 2007]. No entanto, estendemos o meta-modelo com mais um pacote, o qual está destacado em cinza na Figura 4, para podermos adaptá-lo para o conceito de especificação de testes, que é composta por um conjunto de casos de teste.

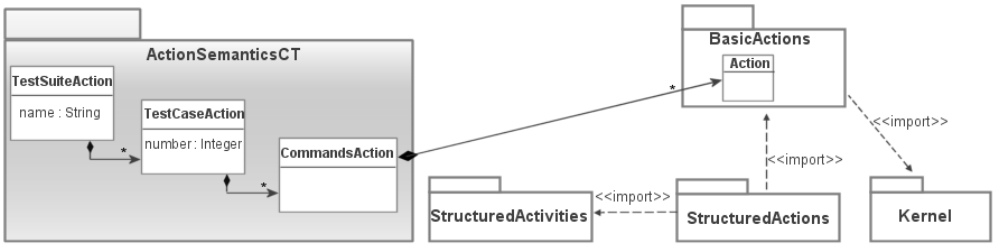

Figura 4: Extensão do meta-modelo de semântica de ações para casos de teste.

Para realizar a transformação dos modelos, foi construído um meta-modelo de saída, que representa uma especificação de testes na linguagem de comandos de USE, onde um caso de teste é composto por um conjunto de comandos da linguagem USE. Esse meta-modelo pode ser visualizado na Figura 5.

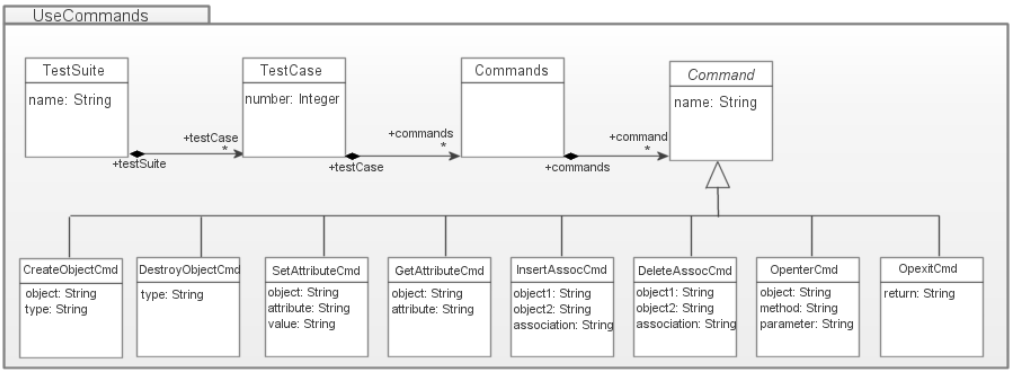

Figura 5: Meta-modelo para os comandos da linguagem de USE.

A Figura 6 apresenta parte das regras de transformação utilizadas. A linha 2 define os meta-modelos de entrada e saída (*ActionSemantics* e *UseCommands*). Na linha 3, temos a definição da regra *createTestSuite,* que cria uma *TestSuite* do meta-modelo *UseCommands* (linha 10) a partir de uma *TestSuiteAction* (linha 5). De acordo com o meta-modelo estendido de *ActionSemantics*, uma *TestSuiteAction* é composta por um conjunto de *TestCaseAction*, que por sua vez é composto por um conjunto de *Action*. Assim, para cada *Action* encontrada no modelo, identifica-se o seu tipo e então é feita uma transformação para o comando de USE equivalente. Por exemplo, na linha 21 temos uma *Action* do tipo *CreateObjectAction* que será transformado (através de uma outra regra ATL) na classe *CreateObjectUseCommand* do meta-modelo de *UseCommands*.

```
1.  module ActionSemantics2UseCommands;
2.  create OUT : UseCommands from IN : ActionSemantics;
3.  rule createTestSuite {
4.    from
5.       tsAs: ActionSemantics!TestSuiteAction
6.    using {
7.       testCasesAc : ActionSemantics!TestCaseAction = tsAs.testCase;
8.    }
9.    to
10.      tsUse : UseCommands!TestSuite(name <- tsAs.name,
11.                 testCase <- tcUse),
12.      tcUse : UseCommands!TestCase(),
13.      cmdsUse : UseCommands!Commands()
14.   do {
15.       for (tcAc in testCasesAc) {
16.          if (tcAc.oclIsTypeOf(ActionSemantics!TestCaseAction))
17.             self.testCaseUse(tcAc, tcUse);
18.             for (cmdsAc in tcAc.commands) {
19.                tcUse.commands <- tcUse.commands->including(cmdsUse);
20.                for (action in cmdsAc.action){
21.                   if (action.oclIsTypeOf(ActionSemantics!CreateObjectAction)) {
22.                       self.createObjectUseCommand(action, cmdsUse);
23.                   }
24.                   ...
25.    } ...
26.  }
```

Figura 6: Regra ATL de transformação de semântica de ações para sintaxe USE.

Após a execução das regras de transformação ATL, obtemos um modelo da linguagem de comandos de USE. No entanto, este modelo ainda não está na sintaxe concreta da linguagem USE, por isso utilizamos a ferramenta MOFScript [Eclipse Project 2006] para transformar o modelo em uma linguagem concreta. Todo o processo MDA para transformação dos casos de teste com semântica de ações em casos de teste em comandos da linguagem USE está descrito na Figura 7.

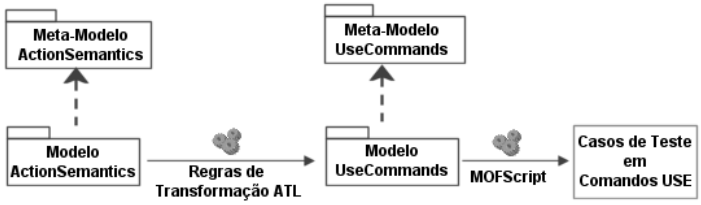

Figura 7: MDA para transformação de semântica de ações em comandos USE.

3.3. Passo 3: Gerando um diagrama de seqüência na ferramenta USE a partir da execução de um caso de teste.

O próximo passo a ser automatizado é a execução dos casos de teste na linguagem USE com a finalidade de verificar a sua validade. Na ferramenta USE, ao executar um comando do caso de teste temos como resultado uma mensagem em um diagrama de seqüência. O diagrama de seqüência gerado será usado como base para a inspeção. É importante notar que um caso de teste pode ser inválido com relação às restrições OCL especificadas no diagrama de classes, ou seja, podem induzir a ações que seriam inválidas durante a verificação das restrições. Neste caso, temos duas possibilidades: i) o caso de teste é inválido e precisa ser revisto, juntamente com o caso de uso correspondente (defeito no documento de requisitos); ii) as restrições OCL anotadas no

diagrama de classes precisam ser revistas (defeito no diagrama de classes). Assim, tanto os casos de teste quanto o próprio modelo estrutural passam por uma revisão neste processo. Serão levados para inspeção apenas os diagramas executados com sucesso.

3.4. Passo 4: Realizando inspeção no diagrama de seqüência de projeto.

Esta etapa corresponde exatamente à etapa de Execução e Avaliação da Inspeção Guiada convencional. Apesar de, na versão atual, a técnica ainda manter a inspeção manual, a diferença é que o processo de automação dos passos anteriores diminui o *gap* semântico entre o caso de teste e o modelo a ser inspecionado. Assim, o processo é bem mais simples e preciso.

Ao inspecionarmos o diagrama de seqüência de projeto, devemos verificar se as mensagens deste diagrama são equivalentes às mensagens do diagrama de seqüência gerado pelo USE. Devemos observar tanto a correspondência entre as mensagens quanto a ordem em que elas aparecem. Se alguma mensagem do diagrama de seqüência inspecionado não estiver em conformidade com as mensagens do diagrama de seqüência base (caso de teste), então devemos adicionar um erro no relatório de defeitos, indicando qual a mensagem daquele diagrama que está com defeito.

4. Exemplo de Aplicação

Nesta seção, apresentamos um exemplo, que utiliza um sistema descrito na Seção 2.4, para ilustrar a automação da Inspeção Guiada e apresentar os resultados obtidos. O exemplo possui a especificação de casos de uso com os requisitos do sistema, o diagrama de classes (apresentado na Seção 2.4) e diagramas de projeto para cada caso de uso. Para esse exemplo vamos ilustrar duas etapas da nossa técnica: (i) transformação e validação dos casos de teste; (ii) inspeção do diagrama de seqüência de projeto. Por restrições de espaço, apresentamos apenas a criação de um caso de teste e a inspeção com este caso de teste em um diagrama de seqüência de projeto. É também importante ressaltar que os diagramas de seqüência de projeto foram criados de forma manual para atender aos requisitos expressos na versão original dos casos de uso.

4.1. Transformação e Validação dos Casos de Teste

Para iniciar a automação da inspeção guiada, criamos os casos de teste abstratos para cada caso de uso do sistema. Logo após, fizemos anotações neles com as respectivas semânticas de ações. A Tabela 4 mostra um dos casos de teste considerados.

Uma vez anotados com semântica de ações, os casos de teste foram transformados automaticamente na linguagem de comandos USE usando MDA. Em seguida, adicionamos o diagrama de classes do sistema (Figura 2) na ferramenta USE. Então, os casos de teste foram executados em USE para gerar automaticamente um diagrama de seqüência. No entanto, ao realizar a primeira execução dos casos de teste, detectamos defeitos relacionados à validação das restrições. Assim, tivemos que identificar se estes defeitos faziam parte da especificação de caso de uso ou do diagrama de classes, pois estes artefatos não estavam em conformidade entre si. Foram encontrados defeitos na especificação de caso de uso relacionados à dependência de criação entre objetos. No diagrama de classes, estas dependências podem ser

visualizadas através da multiplicidade dos relacionamentos entre as classes. Então, a especificação de caso de uso foi corrigida, assim como os casos de teste.

Tabela 4: Caso de teste anotado com as respectivas semânticas de ações.

Caso de Uso 01: Iniciar jogo de xadrez **Dependências:** Adicionar jogadores, Criar partida e Exibir tabuleiro.			**Freqüência:** Alta **Grau de Criticalidade:** Alto
#	**Passos do Caso de Teste Abstrato**	**Passos Anotados com Semântica de Ações**	**Resultado Esperado**
1	1. O sistema inicia um novo jogo de xadrez.	1. `<CreateObjectAction> jogo JogoXadrez` 1.a. `<CallOperationAction> jogo iniciarNovoJogoXadrez()`	O jogo é iniciado com sucesso.
	2. O sistema solicita os nomes dos 2 jogadores.	2. `<CallOperationAction> jogo solicitaNomeJogadores('Jonas','Maria')`	
	3. O usuário informa os nomes dos jogadores.		
	4. O sistema verifica os nomes dos jogadores	4. `<CallOperationAction> jogo vificaNomeValido(nomeJogBr)` 4.a. `<replyAction> true (vificaNomeValido)` 4.b. `<CallOperationAction> jogo vificaNomeValido (nomeJogPt)` 4.c. `<replyAction> true (vificaNomeValido)`	
	5. O sistema adiciona os 2 jogadores no jogo.	5. `<CallOperationAction> jogo adicionaJogador('Jonas', 0)` 5.a. `<CreateObjectAction> jogBr Jogador` 5.b. `<replyAction>> (adicionaJogador)` 5.c. `<CallOperationAction> jogo adicionaJogador('Maria', 0)` 5.d. `<CreateObjectAction> jogPt Jogador` 5.e. `<replyAction> (adicionaJogador)`	
	6. O sistema cria uma partida.	6. `<CallOperationAction> jogo adicionaPartida('100')` 6.a. `<CreateObjectAction> partida Partida`	
	7. O sistema adiciona os jogadores na partida.	7. `<CallOperationAction> jogo adicionaJogadoresPartida(jogBr,JogPt,'100')` 7.a. `<CreateLinkObjectAction> partida jogBr temJogadores` 7.b. `<CreateLinkObjectAction> partida jogPt temJogadores` 7.c. `<replyAction> (adicionaPartida)`	
	8. O sistema exibe o tabuleiro com as peças nas posições iniciais.	8. `<CallOperationAction> jogo exibeTabuleiro()` 8.a. `<CreateObjectAction> tab Tabuleiro` 8.b. `<replyAction> (exibeTabuleiro)` 8.c. `<CallOperationAction> tab exibePecasIniciais()` 8.d. `<replyAction> (exibePecasIniciais)` 8.e. `<replyAction> (iniciarNovoJogoXadrez)`	

4.2. Inspeção dos diagramas de seqüência de projeto

A Figura 8 apresenta o diagrama de sequência gerado automaticamente por nossa técnica através do USE. Este diagrama é resultado da execução do caso de teste apresentado na Tabela 4. Nesse diagrama, podemos observar que as primeiras mensagens: "*create*" e "iniciarNovoJogoXadrez()" são equivalentes aos passos 1 e 1.a da semântica de ações apresentado na Tabela 4. A Figura 9 apresenta o diagrama de seqüência de projeto a ser inspecionado e que foi criado a partir da especificação de caso de uso 01 de forma independente da ferramenta USE. A inspeção foi realizada de forma manual observando a seqüência das mensagens presentes em cada diagrama para verificar se os dois diagramas (o de projeto e o gerado automaticamente através do USE) possuíam o mesmo comportamento. Como resultado da inspeção, encontramos 9 defeitos, os quais foram adicionados ao relatório de defeitos.

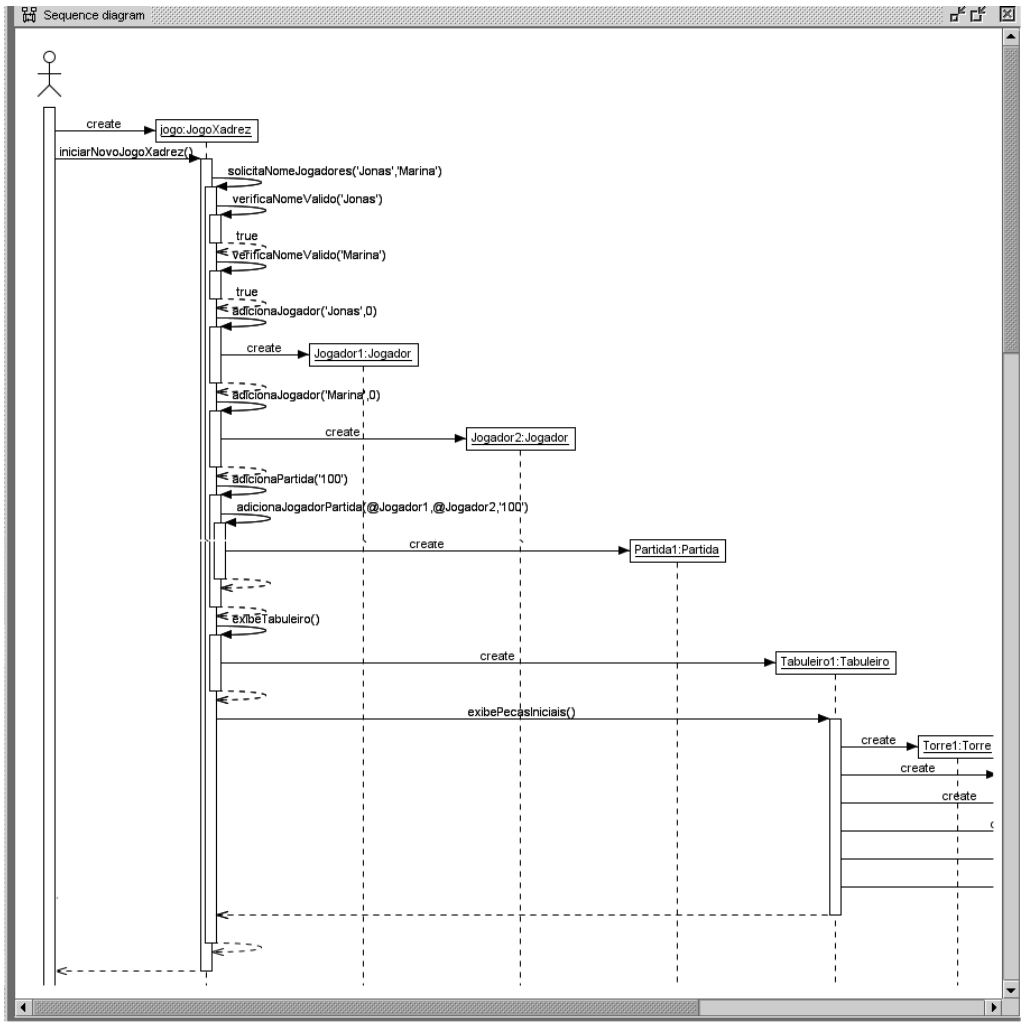

Figura 8: Diagrama de seqüência gerado pela ferramenta USE após a execução do caso de teste.

A Tabela 5 apresenta alguns dos defeitos encontrados, onde cinco defeitos são referentes às mensagens de execução das operações e indicam a qual classe uma determinada operação deveria pertencer (p. e., os defeitos de números 1 e 3). O defeito 2 é referente à regra de negócio, que só permite criar uma partida caso haja 2 jogadores para aquela partida. O defeito 4 é referente à criação de um objeto a partir de uma classe abstrata, que não deve ser permitida. O defeito 5 é referente à falta de detalhe no diagrama de seqüência com relação à criação de cada instância de peça do tabuleiro.

Tabela 5: Relatório de defeitos para o diagrama de seqüência inspecionado.

Relatório de Defeitos – Caso de Uso 01 – Iniciar jogo de xadrez	
1	O método 2: *solicitaNomeJogadores(nomeJogBr, nomeJogPt)* deveria ser executado a partir da classe *JogoXadrez.*
2	Não deve ser possível finalizar a criação de uma partida sem antes ter associado 2 jogadores a ela.
3	O método 3: *exibeTabuleiro()* deveria ser executado a partir da classe *JogoXadrez*, pois não tem como exibir o tabuleiro antes de ele ser criado.
4	As peças devem ser criadas por cada um de seu tipo concreto (Torre, Bispo, Cavalo, etc), pois não há como instanciar uma classe Abstrata (*Peça*).
5	Deveria haver o detalhe sobre a criação das peças. Por exemplo: 2 torres brancas, 2 torres pretas etc.

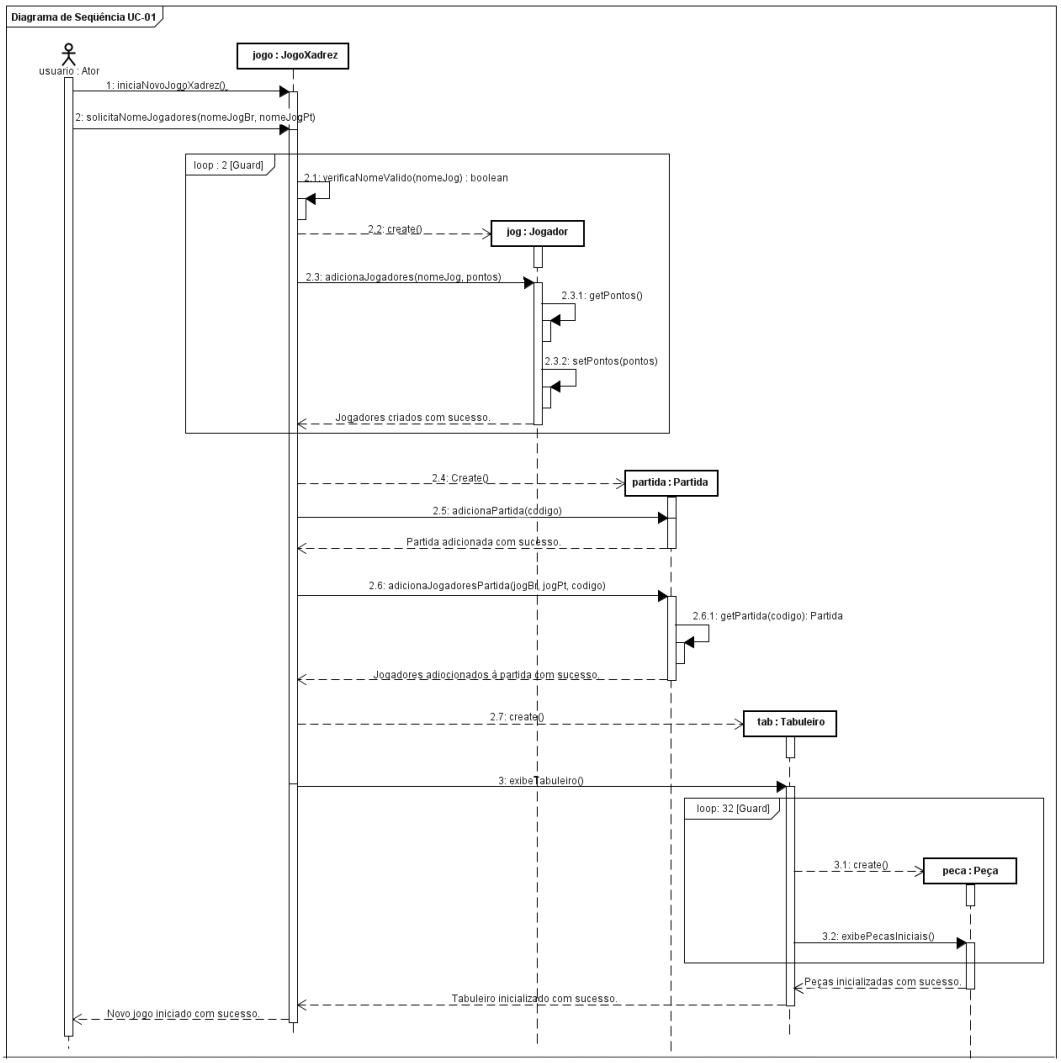

Figura 9: Diagrama de seqüência de projeto para o sistema de xadrez referente ao caso de uso 01.

Como pode ser observado, a inspeção guiada pode trazer grandes benefícios para detecção de defeitos, tanto na especificação base quando no artefato sendo inspecionado. Em um processo de especificação/projeto manual, erros podem ser comumente cometidos. A automação torna o processo mais preciso e confiável, visto que a validação dos casos de teste é fundamental para garantir uma correta interpretação dos resultados.

5. Trabalhos Relacionados

Algumas ferramentas para testar diagramas UML são bem semelhantes à nossa técnica, pois também fazem validação nestes diagramas. No entanto, possuem algumas diferenças com relação ao processo. No caso de ferramentas para inspeção de diagramas UML, elas geralmente dão suporte apenas ao processo tradicional de inspeção, sem haver automação nas regras de inspeção.

Apesar de nossa técnica estar relacionada com inspeção, a ferramenta EPTUD [Trong et al. 2005] para testes em diagramas UML, tem uma abordagem bem semelhante à apresentada neste artigo, mas ela não realiza a Inspeção Guiada. Em EPTUD, a ferramenta UMLAnT [Trong et al. 2005] é utilizada como base para a validação das restrições definidas em OCL (invariantes, pré e pós-condições). No entanto, nessa abordagem é necessário transformar o diagrama de atividades em código executável para que os casos de teste possam ser executados sobre este código. Na nossa técnica, não é necessário fazer nenhuma alteração nos diagramas a serem inspecionados, apenas transformamos os casos de teste em executáveis. Outro ponto a ser observado é que os casos de testes utilizados em EPTUD são gerados automaticamente a partir do diagrama de classes e de diagramas de colaboração, ambos artefatos de nível de projeto. Dessa forma, nesta ferramenta não há uma garantia de conformidade entre os diagramas testados e a especificação de caso de uso, como é feito em nossa técnica. Pois, se faltar algum requisito no diagrama de classes, por exemplo, então os casos de teste gerados também não possuirão este requisito.

Com relação à ferramenta de suporte ao processo de inspeção de diagramas UML [Ohgame e Hazeyama 2006], ela não faz Inspeção Guiada, mas garante um processo de inspeção eficiente e organizado. Além disso, permite fazer controle de versão de cada artefato e também delegar as tarefas para cada membro da inspeção. No entanto, a inspeção é realizada de forma manual utilizando uma lista de instruções pré-definidas. Os defeitos encontrados nos diagramas são adicionados através de comentários, no mesmo formato definido para UML, o que facilita a visualização dos defeitos pelos analistas durante a correção.

6. Conclusões

Apresentamos uma técnica de automação da técnica de Inspeção Guiada que realiza inspeção em diagramas UML utilizando casos de teste. Durante a automação, utilizamos o padrão de Semântica de Ações de UML 2, definido pelo OMG, como também usamos técnicas de MDA para transformação de modelos e a ferramenta USE para simulação de modelos. Apesar da técnica de Inspeção Guiada convencional realizar a inspeção entre artefatos de diferentes níveis de abstração, nossa técnica preservou estas características, mesmo fazendo inspeção entre artefatos de mesmo nível de abstração. Com a técnica de MDA juntamente com a ferramenta USE, foi possível transformar artefatos de requisitos para o nível de projeto, de modo a facilitar a inspeção. Um ponto importante a ressaltar é que esta transformação não tem como objetivo substituir o processo manual de projeto, já que os casos de teste seriam representações fiéis dos casos de uso. Os casos de teste executáveis têm o único propósito de inspecionar artefatos de projeto que devem ser feitos de forma criativa tal como demanda esta etapa, com base em outros fatores tais como requisitos não-funcionais que não são considerados em nossa técnica. No momento, a técnica proposta está limitada a realizar inspeção em diagramas de seqüência.

Com relação à ferramenta USE, não foi trivial definir as invariantes, pré e pós-condições, pois é necessário um amplo conhecimento de OCL (*Object Constraint Language*). Além disso, os diagramas de seqüência gerados por USE não dão suporte a UML2, logo não possuem, por exemplo, fragmentos combinados de “loop”, “ref”, “alt”, etc, o que pode requerer um pouco mais de atenção durante a inspeção.

A técnica de Inspeção Guiada apresentada é válida, pois com ela é possível detectar defeitos tanto nos diagramas de projeto, como também na especificação de requisitos. Como trabalhos futuros, pretendemos realizar a inspeção entre os diagramas de seqüência de forma automática, também adotando técnicas MDA, e assim gerar um relatório de defeitos, que pode ser apresentado em UML ou OCL.

Referências

ATL Project. (2008) "The ATL User Manual". Disponível em: http://www.eclipse.org/m2m/atl/doc/ Acessado em: 20/05/2009.

Eclipse Project. (2006) "MOFScript User Guide". Disponível em:http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf Acessado em: 08/06/2009.

Gogolla M., Büttner F., e Richters, M. (2007). " USE: A UML-Based Specification Environment for Validating UML and OCL". Sci. of Comp. Programming, 69:27-34.

Ho, W. M., Jquel, J.-M., Guennec, A. L., e Pennaneac'h, F. (1999). "UMLAUT: An extendible UML transformation framework". Automated Soft. Eng., 275–278.

Hutcheson, M. L. (2003). Software Testing Fundamentals: Methods and Metrics. John Wiley & Sons, Inc., New York, NY, USA 1st edition.

Jorgensen, P. C. (1995). Software Testing: A Craftsman's Approach. CRC Press, Inc., Boca Raton, FL, USA, 1st edition.

Kamperman, J. (2003). "Automated software inspection: A new approach to increased software quality and productivite". Reasoning Inc. Disponível em:http://www.apacheweek.com/issues/ Acessado em: 10/06/2009.

Kleppe, A., Warmer, J., e Bast, W. (2003). MDA Explained: The Model Driven Architecture: Practice and Promise. Addison Wesley, 1st edition.

Major, M. L. e McGregor, J. D. (1999). "Using guided inspection to validate umlmodels". In: 25th annual Software Engineering Workshop, SEW99.

McGregor, J. D. e Sykes, D. A. (2001). A Practical Guide to Testing Object-Oriented Software. Object Technology Series. Addison-Wesley, 2 nd edition.

OMG, Object Management Group (2005). "Ocl 2.0 specification". Technical Report ptc/2005-06-06, OMG. Disponível em: http://www.omg.org/docs/ptc/05-06-06.pdf Acessado em: 10/05/2009.

OMG, Object Management Group (2007). "Uml superstructure, v2.1.1". Technical Report formal/07-02-05, OMG. Disponível em: http://www.omg.org/cgi-bin/doc?formal/07-02-05 Acessado em: 10/05/2009.

Sommerville, I. (2007). Software Engineering. Addison-Wesley, United Kingdom, 8th edition.

Trong, T. D., Kawane, N., Ghosh, S., France, R., e Andrews, A. (2005). "A Tool-Supported Approach to Testing UML Design Models". In Proceedings of the ICECCS'05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems.