3. Estado da Arte

Este capítulo apresenta os principais trabalhos correlatos encontrados na literatura. A seção 3.1 apresenta um panorama dos trabalhos relacionados à *geração de testes baseada em modelos*, mas com um enfoque diferente do presente trabalho. A seção 3.2 apresenta um panorama dos trabalhos com o mesmo enfoque do presente trabalho. Por fim, a seção 3.3 compara os trabalhos, mostrando as principais influências, diferenças e restrições.

3.1.Trabalhos parcialmente correlatos

A pesquisa realizada teve foco em trabalhos que usam a **descrição textual de casos de uso** como *principal* origem para a geração de testes. Neste enfoque, foram **desconsiderados** para detalhamento os seguintes tipos de trabalho:

- Que usam diagramas UML (OBJECT MANAGEMENT GROUP, 1997) como principal fonte para geração de testes;
- Que fazem a especificação de casos de uso de maneira formal, usando linguagens como Z (MEYER & BAUDOIN, 1980);
- Que se concentram na análise e geração de mutantes para testes;
- Que descrevem casos de uso usando OCL (OBJECT MANAGEMENT GROUP, 2003) ou qualquer outro meio de representação diferente de linguagem natural;
- Que derivam casos de teste de maneira não automática, com criação manual de tabelas de decisão, grafos de causa-efeito ou artefatos similares;
- Que se concentram unicamente em testes de interface com o usuário sem considerar a especificação de requisitos.

Apesar de estas abordagens poderem gerar bons resultados, elas, em sua grande maioria, não especificam requisitos funcionais ou não criam meios de verificar se

o software construído corresponde a esses requisitos. Quando criam meios, a especificação não é escrita numa linguagem fácil de ser compreendida por usuários leigos, requerendo treinamento. Este é um dos motivos que nos levou a usar casos de usos. Casos de uso são fáceis de entender e criar e possibilitam gerar testes funcionais para verificar a qualidade do software construído.

A Tabela 4, a seguir, apresenta os principais trabalhos encontrados na área de testes baseados em modelos (*model-based testing*) que não utilizam a descrição textual de casos de uso como a *principal* fonte para a geração de testes. A tabela apresenta as abordagens escolhidas em cada trabalho, permitindo compará-las e fornecendo uma visão geral da área. As colunas da tabela possuem as seguintes abreviações das abordagens utilizadas:

- **AC**: Análise de Código;
- **EF**: Especificação Formal;
- **EE**: Especificação de Estados;
- **UML**: *Unified Modeling Language*;
- **DT**: Descrição Textual de casos de uso;
- **TE**: Teste de Mutação;
- **EEI**: Especificação de Eventos de Interface;
- **DD**: Dicionário de Dados;
- **OCL**: *Object Constraint Language*;
- **IC**: Instrumentação de Código;
- **DSL**: *Domain Specific Language*;
- **GV**: Geração de Valores para testes.

Tabela 4 - Enfoque de trabalhos parcialmente correlatos

Trabalho	AC	EF	EE	UML	DT	TM	EEI	DD	OCL	IC	DSL
Ryser & Glinz (1999)			X	X							
Ramakrishnaiah (1999)	X	X									
Maldonado *et al.* (1999)			X	X		X					

Trabalho	AC	EF	EE	UML	DT	TM	EEI	DD	OCL	IC	DSL
Chernonozhkin (2000)	X										
Memon *et al.* (2001)							X				
Lebiche & Briand (2002)				X				X	X		
Chen & Subramaniam (2002)			X								
Memon *et al.* (2003)							X				
Barnett *et al.* (2003)			X							X	
Dinh-Trong (2004)				X					X		
Okum (2004)						X					
Sinha (2005)			X								X
Memon & Zie (2005)							X				
Dinh-Trong (2006)				X							X
Nebut *et al.* (2006)				X					X		
Samuel (2007)		X	X								
Memon & Zie (2007)							X				
Chen (2007)				X						X	
Briand *et al.* (2008)				X					X		
Im *et al.* (2008)											X
Yuan (2008)							X				
Ran *et al.* (2011)			X	X				X			
Xie *et al.* (2009)											X
Navarro *et al.* (2010)							X		X		
Yuan & Memon (2010)							X				
Jaygarl (2010)	X										
El-Attar & Miller (2010)					X		X			X	
Hassan & Yousif (2010)				X	X						
TOTAL	**3**	**2**	**7**	**10**	**2**	**2**	**8**	**2**	**5**	**3**	**4**

Como podemos observar, o uso da UML, a especificação de estados, a especificação de eventos de interface e o uso da OCL são as abordagens

encontradas mais utilizadas fora do contexto da geração a partir de casos de uso.

A seguir, esses trabalhos são resumidos, como forma de fornecer um panorama sobre a geração de testes baseados em modelos.

- Ryser & Glinz (1999) mapeiam cenários de casos de uso com Diagramas de Estado (UML), num processo manual;
- Ramakrishnaiah (1999) geram testes unitários a partir de uma especificação formal e da análise estática do código, num processo automático;[6,7]
- Maldonado *et al.* (1999) usam teste de mutação para validar especificações baseadas em Diagramas de Estado (UML),[8] num processo semiautomático;
- Chernonozhkin (2000) gera testes unitários a partir da análise estática do código, num processo automático;
- Memon *et al.* (2001) estabelecem critérios de cobertura para testes de interface com o usuário, baseados em seus eventos e interações.[9] O trabalho não considera os requisitos do *software* a ser testado;
- Labiche & Briand (2002) realizam a geração de testes a partir dos diagramas UML de Casos de Uso, Sequência (para cada Caso de Uso), Classes (com classes do domínio da aplicação), além da Descrição Textual de Casos de Uso, de um Dicionário de Dados (que descreve cada classe, método e atributo), e da OCL, para descrever Pré-condições e Pós-condições;
- Chen & Subramaniam (2002) utilizam um editor de estados se baseia no uso de Máquina de Estados Finitos para a geração dos casos de teste. A edição dos estados, a geração dos testes e a escolha dos valores dos testes são realizadas de forma manual. A ferramenta criada, chamada de VESP, gera testes para AWT[10] e Swing;[11]

[6] O trabalho apresenta uma boa discussão sobre análise de valores limite, para geração de valores para testes.
[7] Uma ferramenta, chamada de Apollo, é criada pelo trabalho.
[8] Uma ferramenta, chamada de PROTEUM/ST, é criada pelo trabalho.
[9] Uma ferramenta desenvolvida num trabalho anterior, chamada de PATHS, auxilia na geração de casos de teste analisados no trabalho.
[10] *Abstract Window Toolkit*. http://docs.oracle.com/javase/6/docs/technotes/guides/awt/

- Memon *et al.* (2003), que continuam do trabalho de 2001 (ver acima), se concentram na criação de testes rápidos para interface com o usuário (*smoke tests*), com frequência diária. A ferramenta criada, chamada de DART, inspeciona a interface com o usuário fazendo a extração (GUI *ripping*) de seus elementos, propriedades e transições. Com a ferramenta, o testador inspeciona visualmente a estrutura da IU extraída e faz correções para que entre em conformidade com a especificação (que não é abordada pela ferramenta). Então, testes são gerados e é feita uma análise da cobertura do código, eventos e interações da IU. A cobertura de eventos leva em conta se cada combinação possível é verificada por pelo menos um caso de teste. A cobertura de código, construída para Java, detecta os *listeners* dos eventos dos elementos da interface e anexa *listeners* próprios, que registram o recebimento do evento. O executor de testes dispara todos os eventos identificados, sendo um em cada caso de testes, e compara a saída obtida com a saída esperada. Porém, a ferramenta não leva em conta a especificação do *software*, não gera valores para testes e não tenta explorar possíveis falhas decorrentes de entradas inválidas. Além disto, para o objetivo proposto (testes rápidos e diários), a ferramenta se mostrou lenta. Para apenas um *software* analisado, a realização dos testes durou aproximadamente 52 horas;

- Barnett *et al.* (2003) desenvolveram a *Abstract State Machine Language* (AsmL), integrada ao *framework* .NET da Microsoft,[12] para automação de testes. O ambiente criado usa uma abordagem semiautomática, requerendo que o usuário faça anotações sobre o modelo da aplicação, para gerar parâmetros e sequências de chamadas e configurar as amarrações entre o modelo e a implementação. Destas anotações é gerada uma Máquina de Estados Finitos e então um Gerador de Sequências deriva os testes da MEF. Por fim um Verificador em Tempo de Execução testa se a implementação está de acordo com as anotações criadas;

[11] http://docs.oracle.com/javase/6/docs/technotes/guides/swing/
[12] http://msdn.microsoft.com/pt-br/netframework/

- Dinh-Trong (2004) propõe uma abordagem sistemática para testar modelos de projeto criados com os diagramas de Classes, Sequência e Atividades (ambos da UML). As Pré-condições e Pós-condições são descritas usando a OCL. O processo apresentado é manual e uma ferramenta seria desenvolvida como trabalho futuro;

- Okum (2004) se concentra na especificação de mutação para análise e geração de testes;

- Sinha (2005) gera testes baseados numa especificação construída por uma linguagem de domínio específico (*Domain Specific Language*, DSL) de alta ordem. Para isto é usada a linguagem HaskellDB e seu código é transformado numa representação intermediária similar à uma Máquina de Estados Finitos. Esta representação pode servir como entrada para outras ferramentas que geram testes baseados em MEF;

- Memon & Xie (2005), que continuam o trabalho de 2003 (ver acima), usam a ferramenta DART para automação de testes rápidos de regressão (*smoke regression tests)* para interface com o usuário. Os autores concluem que: o processo de testes proposto é viável em termos de tempo de execução, espaço de armazenamento e esforço manual; *smoke tests* podem não cobrir certas partes da aplicação; ter oráculos compreensivos (o conceito de "compreensivo" não é definido no trabalho) pode compensar a ausência de casos de teste longos (que são percebidos como melhores); usar certos oráculos (o trabalho não define bem quais os tipos de oráculos devem ser usados) pode compensar a ausência de suítes de teste grandes (que são percebidas como melhores);

- Dihn-Trong (2006), em continuação ao trabalho de 2004 (ver acima), cria a ferramenta UMLAnT e uma linguagem de especificação, chamada de JALS (*Java like Action Language Specification*), usada em substituição à OCL;

- Nebut *et al.* (2006) se baseiam: em Diagramas de Sequência (UML); na descrição de Pré-condições e Pós-condições em forma de expressões lógicas (feitas em um editor próprio); e na descrição de contratos sobre

classes, especificados em OCL. Os autores construíram uma ferramenta, chamada de UC-System, para implementar o processo definido;

- Samuel *et al.* (2007) geram testes a partir de diagramas de Estados (UML), com apoio da ferramenta criada, chamada de UML *Test Case Generator*;
- Memon & Xie (2007), em continuação ao trabalho de 2005 (ver acima), se concentram nos tipos de oráculo e na efetividade de cada um (o que não tinha sido bem detalhado no trabalho anterior). É afirmado que os oráculos afetam a habilidade de detectar falhas e que o tamanho dos casos de teste influencia em sua habilidade de detecção de falhas (este último a ser investigado num trabalho futuro);
- Chen *et al.* (2007) geram casos de teste a partir de Diagramas de Atividade (UML) e instrumentação de código Java, de acordo com os diagramas;
- Briand *et al.* (2008), em continuação do trabalho de 2002 (ver acima), gera casos de teste para Diagramas de Sequência (UML) e compara o diagrama e o código que o implementa, detectando diferenças. O trabalho usa XML e OCL e realiza estudos de caso com exemplos acadêmicos e da indústria. Como trabalho futuro, comparará seus resultados com os trabalhos que se baseiam na análise do código;
- Im *et al.* (2008) usa OWL[13] como uma DSL para especificar casos de uso e gera testes a partir desta especificação;
- Yuan (2008), que dá sequência ao trabalho de Memon & Zie (2007), particiona os eventos de interface em grupos para testar sua interação. Esta divisão, assim como nos trabalhos anteriores, é realizada manualmente. Ou seja, o usuário deve usar seu conhecimento de domínio do negócio para determinar que eventos possuem funcionalidades relacionadas. O autor informa que esta automatização deverá ser feita em trabalhos futuros. Percebe-se, também, que o tempo total de geração e execução dos testes continua lento. Assim como nos trabalhos anterio-

[13] *Web Ontology Language*. http://www.w3.org/TR/owl-ref/

res, o teste de uma aplicação selecionada (TerpOffice Paint) num aglomerado (*cluster*) de 50 computadores durou 3,5 dias para os testes rápidos e 14 dias para os testes completos. Além disto, a cobertura máxima alcançada foi de aproximadamente 78% e houve uma grande quantidade de testes gerados cuja execução obteve erro (testes incorretos). Para um grupo de testes executado (Grupo 5), por exemplo, 96% dos testes gerados não conseguiram completar sua execução;

- Ran *et al.* (2008) se concentram nos testes de bancos de dados de aplicações *web*, gerando casos de teste funcionais que levam em conta atualizações no banco de dados da aplicação alvo. A ferramenta construída, chamada de *Automatic Database Tester*, usa: um Diagrama de Estados (UML) que mostra como o usuário navega nas páginas do *site*; uma Especificação de Dados, que mostra como os dados fluem entre as páginas; uma Especificação de Atualização, que mostra como o banco de dados é atualizado após uma interação com uma página; e um Grafo de Herança de Dados, para especificar a dependência de dados entre páginas. A duas especificações citadas são escritas em Prolog.[14] A ferramenta gera verificações que realizam consultas para determinar o estado do banco de dados e oráculos parciais, que ajudam a verificar se o banco de dados foi corretamente atualizado durante a execução dos testes. Modificações estruturais no esquema (*schema*) do banco de dados não foram consideradas;

- Xie *et al.* (2009) fazem a geração de valores para testes baseado no fornecimento de expressões regulares. O trabalho gera uma implementação para C#, potencialmente portátil para outras linguagens. O trabalho informa ter alcançado 79% de cobertura, melhorando em 29% em relação ao seu não uso;

- Navarro *et al.* (2010) se concentram em validar a interface com o usuário, sem confrontar com a especificação. Para definir as restrições dos campos da interface com o usuário (*widgets*), propõem o uso de OCL ou de Ruby (o testador precisa conhecer estas linguagens). Após definir

[14] http://www.swi-prolog.org/

Pré-condições, Pós-condições e Invariantes para cada *widget* da interface, o testador usa a ferramenta construída pelo trabalho para gerar testes que verificam todas as combinações entre os valores dos *widgets*. Como o número de combinações pode ser muito alto, ele pode inviabilizar a aplicação prática da solução. O trabalho também define um algoritmo para a geração de casos de teste e cria a verificação para as propriedades dos *widgets* de forma dependente de plataforma (para uma determinada linguagem e biblioteca de código). Os autores criaram um *widget adapter* para poder obter as propriedades de cada *widget* como uma *string*, para facilitar a comparação, mas não definiram se este adaptador é construído pelo testador ou se é embutido na ferramenta. Os valores usados nos testes não são gerados pela ferramenta, mas definidos manualmente pelo testador;

- Yuan & Memon (2010) dão prosseguimento ao trabalho de 2008 (ver acima) e fazem uma boa revisal de trabalhos anteriores, aprofundando seu estudo sobre testes de interações entre eventos de interface com usuário, que é baseado na análise de *feedback* obtido do estado dos *widgets* em tempo de execução. **O estudo mostrou que a efetividade dos testes obtida foi devida à identificação dos relacionamentos entre os elementos da interface e não ao tamanho da suíte de testes, nem dos eventos adicionais ou do código que ele cobre**;
- Jaygarl (2010) explora a geração de testes a partir da análise estática e dinâmica (em tempo de execução) do código. A ferramenta gerada, chamada de CAPTIG, captura as entradas usadas durante a execução do software, além de gerar entradas aleatórias para os testes. Ela analisa caminhos não percorridos e faz mutação das entradas para tentar cobri-los, baseado na análise da pré-condição, o que auxilia a obter maior cobertura. Baseada no rastro da pilha (*stack*), também consegue reproduzir travamentos. O trabalho compara seus resultados com os de uma ferramenta de testes aleatórios, obtendo desempenho de 5% a 22.8% melhor, com entradas de dados e tempo bem inferiores (o que é melhor). Nos experimentos conduzidos, consegue-se reproduzir 44.6% dos travamentos;

- El-Attar & Miller (2010) se concentram em testes de aceitação, fazendo uso da Descrição Textual de Casos de Uso (DTCU), de Diagramas de Classes e de Robusteza (UML), além de uma linguagem criada pelos autores, similar à OCL, para expressar as restrições da DTCU. Somente os fluxos Básicos e Alternativos da DTCU são utilizados. Sua ferramenta, chamada de UCAT, faz uso das ferramentas FIT[15] e Fitnesse, [16] onde a DTCU criada é aumentada com o uso dos *fixtures*[17] (do FIT). O trabalho não mostra os processos e algoritmos usados. As ações que o sistema deve executar se resumem a exibir as respetivas visões esperadas, correspondentes às classes e métodos. Os dados de teste não são gerados e o estudo de caso apresentado é pouco abrangente e desprovido de métricas;

- Hassan & Yousif (2010) geram grafos de atividades a partir dos passos da descrição textual de casos de uso. Cada nó num grafo pode ser inicial, final, de decisão, de guarda, ou de ação. A partir desses grafos são gerados cenários, usando busca em profundidade (*Depth First Search*). Como critério de cobertura, eles usaram o que chamaram de "cobertura de caminho de ação", que cobrem ramos e *loops*. A ideia é que cada ação seja executada por um teste, pelo menos uma vez. O percentual de cobertura é calculado dividindo-se o número de nós de ação do cenário pelo número total de nós de ação do caso de uso. O trabalho, porém, só gera cenários, não gerando testes, dados de teste, ou código de teste;

- Isabella & Retna (2012) fazem um bom levantamento sobre trabalhos sobre teste para interfaces com o usuário, mas sem foco nos requisitos;

Diversos outros trabalhos foram omitidos, principalmente aqueles relacionados à geração de casos de teste e oráculos baseado em mutantes, geração de mutantes, algoritmos para geração de casos de teste baseada em modelos e análise de cobertura do código.

[15] Disponível em: http://fit.c2.com/
[16] Disponível em: http://fitnesse.org/
[17] No Fit, um *fixture* é um código criado pelo programador para entender a estrutura da tabela usada nos testes e como ela será interpretada pelo programa. A ferramenta Fit usa tabelas com exemplos de como o programa deve funcionar, podendo o usuário não técnico usá-las para entrar com dados de teste.

3.2.Principais trabalhos correlatos

A principal característica pesquisada foi a especificação de requisitos através de casos de uso em linguagem natural, com possível uso de vocabulário restrito e semiestruturado, e com geração de cenários e casos de teste baseada nesta especificação. Os principais trabalhos encontrados são apresentados a seguir. Adiante, na Tabela 6, as ferramentas neles produzidas são comparadas e na seção 3.3 é realizado um comparativo em relação ao presente trabalho.

Heumann (2001) apresenta um resumo sobre a descrição textual de casos de uso, descreve como gerar cenários a partir deles e como compor casos de teste a partir destes cenários. O processo é explicado através de exemplos e numa abordagem informal. A automatização do processo não é discutida. As ideias do artigo são boas, porém o assunto não é aprofundado, faltando detalhar vários pontos importantes. Seus pontos de destaque são:

- A identificação dos cenários, através das combinações possíveis dos fluxos criados;
- O estabelecimento de um processo de três passos para a geração de casos de teste:
    1. Para cada caso de uso, gerar um conjunto completo de cenários;
    2. Para cada cenário, identificar pelo menos um caso de teste e as condições que o farão "executar";
    3. Para cada caso de teste, identificar os valores para uso nos testes.

Denger & Mora (2003) fazem uma extensa análise do estado da arte, que é analisada posteriormente por Gutiérrez (2005), Gutiérrez *et al*. (2005), Gutiérrez *et al*. (2010) e por Escalona *et al.* (2011). Este último também resume os trabalhos anteriores. Dele podemos destacar:

- A listagem dos aspectos resolvidos e não resolvidos sobre o assunto, que são:

1. Aspectos resolvidos:

   - *Construção do modelo comportamental*, que permite sistematizar e automatizar o processo. Geralmente é usada a abordagem de Testes Baseados em Modelos.
   - *Ferramentas de suporte*, que geralmente são construídas nos trabalhos apresentados, o que mostra um grau de maturidade e uma melhoria da automação. Porém, muitas das ferramentas não são disponibilizadas ou não possuem código-fonte aberto;
   - *Usar dois níveis na derivação de casos de teste*, que é usado em vários trabalhos, mas de diferentes formas. Num nível é testado o caso de uso isoladamente. No outro, é testado em conjunto com outros casos de uso e é verificado o atendimento aos requisitos funcionais;
   - *Definição dos requisitos*, que é realizada de várias formas (ex.: há vários modelos de casos de uso), mas todos indicam os detalhes da abordagem escolhida.

2. Aspectos não resolvidos:

   - *Documentação e apresentação de casos práticos*, que são, ambas, escassas. Sua falta prejudica a avaliação da real aplicação das abordagens;
   - *Inexistência de medida de efetividade e qualidade*, que é esquecida por todas as abordagens. Todas pressupõem que a cobertura máxima é alcançada, mas não aplicam nenhuma métrica para avaliá-la. Nenhuma demonstrou, com experimentos, se sua abordagem é melhor que o uso de testes aleatórios ou que o uso do senso comum;
   - *Falta de sistematização e automação*;
   - *Implementação de casos de teste*, que são omitidas na maioria dos trabalhos;
   - *Falta de estudos empíricos*.

- Uma lista de tarefas proposta para gerar casos de teste a partir de casos de uso, baseada na listagem dos aspectos resolvidos e não resolvidos:

    1. Construir o modelo comportamental;
    2. Derivar cenários de teste de um caso de uso;
    3. Derivar cenários de teste de vários casos de uso;
    4. Gerar os valores de teste;
    5. Obter cenários de teste;
    6. Reduzir o número de casos de teste sem perda da cobertura;
    7. Medir a cobertura;
    8. Gerar resultados esperados;
    9. Ordenar casos de teste para maximizar o critério de seleção (priorização);
    10. Gerar *scripts* de teste ou executar o código de teste.

A principal conclusão do trabalho é que há abordagens suficientes para adquirir ideias precisas de como derivar casos de teste. Contudo, ainda não há uma abordagem completa e integrada que descreva todo o processo.

Kassel (2006) descreve uma abordagem para a geração automática de casos de teste a partir de casos de uso, onde casos de uso são documentados no formato XML (WORLD WIDE WEB CONSORTIUM, 1998), num *template* chamado pelo autor de *Use Case Mark-up Language* (UCML). Sua descrição de caso de uso inclui:

- Nome do Caso de Uso
- Descrição Breve
- Fluxo de Eventos
- Pré-condições e Pós-condições
- Criticidade
- Frequência de Uso

Cada fluxo de eventos é divido em passos (como é de costume na descrição de fluxos) e cada passo é dividido em quatro partes:

1. *Nome do ator*: identificação de quem executa a ação;
2. *Ações do ator*: identificação da ação do ator ou de uma verificação do sistema;
3. *Nome do objeto*: Identificação do objeto a ser testado;
4. *Atributo do objeto*: Especificação do dado do objeto que será usado no teste.

As Pré-condições e Pós-condições são usadas tanto para casos de uso quanto para os Fluxos de Eventos. Para estruturá-las, é feito uma divisão em três partes:

1. *Nome do objeto*;
2. *Atributo do objeto*;
3. *Valor*;

Seu uso nos Fluxos de Eventos é justificado pela afirmação do uso de fluxos alternativos de exceções terminadas de forma normal ou anormal e, para cada uma, deve haver pós-condições que verifiquem determinados estados.

A Criticidade é a importância do caso de uso para o usuário do sistema, definida como "baixa", "média" ou "alta" e servirá como um peso do caso de uso para o sistema, na hora da priorização dos casos de uso para a geração dos testes. Ela será usada junto à Frequência de Uso, que estabelece a frequência de uso num caso de uso numa escala de tempo, para calcular a prioridade do caso de uso e assim estabelecer sua ordem na geração dos testes.

Também é proposto no trabalho um conjunto de passos para automatizar a geração dos testes:

1. Para cada caso de uso, gerar um conjunto completo de cenários;
2. Calcular a prioridade de cada caso de uso;
3. Gerar uma máquina de estado a partir das pré-condições e pós-condições dos casos de uso;

4. Usar o resultado dos passos 1 a 3 para criar sequências de teste dos estados iniciais aos finais;

5. Para cada sequência de teste, identificar pelo menos um caso de teste e elementos que irão fazê-lo "executar";

6. Para cada caso de teste, identificar os dados que participarão do teste.

Estes passos são detalhados e usados pela ferramenta construída para a geração dos testes.

É importante observar que na ferramenta construída:

- ➔ Não é permitida a confecção de casos de uso através de uma interface com o usuário. É preciso editar manualmente o arquivo XML (em UCML) para confeccionar os casos de uso;
- ➔ Não é realizada a geração automática dos valores usados nos testes. Após a geração dos casos de teste, o usuário precisa editar manualmente o arquivo XML e inserir os valores a serem usados.
- ➔ Não é abordada a geração de cenários que envolvem mais de um caso de uso.

O trabalho determina métricas de cobertura para os testes gerados, usando:

- *Cobertura de Passo*: Cada passo deve ser usado;
- *Cobertura de Cenário*: Cada fluxo tem de ser usado;
- *Cobertura de Ramo*: Cada seleção de uma lista de opções (ex.: *combo*) deve ser testada; E cada passo opcional deve ser pulado pelo menos uma vez e considerado pelo menos uma vez.

O tempo de execução obtido é da ordem de $O(n^3)$, sendo n o número de estados da Máquina de Estados utilizada. Apesar da taxa de crescimento do algoritmo usada na ME ser menor que $O(n^3)$, os tempos de execução incluem todo o procedimento de geração do código, não só os algoritmos. A geração da sequência de testes, por exemplo, incluem uma mistura de código cujos algoritmos variam de $O(n^2)$ a $O(n^3)$, como a gravação de arquivos XML e a visualização dos dados na tela.

Gutiérrez *et al.* (2008) descrevem um estudo de caso que verifica a correta implementação de um sistema *web* e um sistema de linha de comando, medindo e analisando a efetividade da técnica de análise de cenários, que permite gerar casos de teste a partir de casos de uso. O estudo de caso não somente gera os casos de teste, mas os implementa e os executa para avaliar sua efetividade.

Primeiramente os casos de teste dos sistemas são descritos num arquivo XML, cujas marcações acomodam a descrição textual do caso de uso com seus campos usuais. Deste arquivo XML é realizado um processamento para gerar um Diagrama de Atividade (DA) que representa os passos de seus fluxos. Para gerar os cenários a partir do DA, são implementados os seguintes critérios de seleção:

- *Todos os Nós*: Seleciona os caminhos que passam por um maior número de ações do DA, até que todas as ações do diagrama tenham sido percorridas pelo menos uma vez;
- *Todas as Transições*: Seleciona o caminho que passa pelo maior número de vértices de fluxo, até que todos tenham sido percorridos pelo menos uma vez;
- *Todos os Cenários*: Se o DA não entrar em *loop*, seleciona os caminhos que passam por todas as saídas dos vértices de fluxo, dos nós de decisão, pelo menos uma vez. Se tiver alguns *loops*, são selecionados os caminhos que passam por todas as saídas dos vértices de fluxo e todas as combinações envolvendo os *loops* pelo menos uma vez.

Para medir a efetividade dos casos de teste gerados são utilizados *mutantes*, onde o número de mutantes mortos (que significa que um comportamento diferente do esperado foi detectado) é usado como fator medido. Entretanto, o uso de mutantes empregado não foi o convencional, usando *operadores mutantes* no código para que ele se comporte diferente e os testes gerados consigam capturar esta mudança de comportamento. Ao invés disso, propôs-se usar *casos de uso mutantes*, que descrevem um comportamento diferente para o caso de uso. Para isto, foi definido um novo conjunto de operadores mutantes, do modelo de falhas de caso de uso introduzidas por Binder (2000), exibido na Tabela 5, a seguir.

Tabela 5 - Operadores mutantes para casos de uso

#	Operador mutante
1	Troca de operador lógico da condição de uma alternativa ou a substituição de um passo com erro
2	Troca da condição de uma alternativa ou de um passo com erro sempre avaliado como *true* ou *false*
3	Um passo terminado de repente
4	A remoção de um passo
5	A adição de um passo
6	Substituição de regras de validação ou da admissão de um dado como incorreto
7	Informação incorreta ou incompleta mostrada pelo sistema
8	Uma operação que pode falhar, mas sempre funciona corretamente
9	Uma operação com falha sem ter um passo incorreto
10	A informação mostrada para o ator tem menos elementos
11	Substituição do executor do passo

Foram usados, em média, 23 mutantes para o sistema *web* e 17,7 mutantes para o sistema de linha de comando, que possuem respectivamente 4 e 3 casos de uso.

A abordagem *Todos os Cenários* obteve efetividade de 82,6% para o sistema *web* e de 84,9% para o de linha de comando. A abordagem *Todos os Nós* obteve respectivamente 65,2% e 84,9%. E a abordagem *Todas as Transições* obteve respectivamente 78,3% e 84,9%.

O trabalho relata que era esperada da abordagem *Todos os Nós* uma maior efetividade, devido ao número de casos cobertos ser maior. Entretanto, foi justificado que o resultado obtido ocorreu devido ao uso do *framework* Struts (na linguagem Java) que, segundo os autores, impõe uma maneira rígida de trabalhar, com *cache* de objetos, mecanismos de sessão, etc. Segundo relatam, isto fez com que, se uma operação fosse executada com sucesso da primeira vez, da segunda em diante ela não seria repetida, devido ao *cache*.

O trabalho conclui que, apesar de alguns estudos apontarem que as falhas encontradas em sistemas reais são menos elaboradas que as criadas por mutantes,

o uso de mutantes pode ser valioso para a análise de cenários desses sistemas; que estudos de caso e experimentos são difíceis de realizar devido à geração e implementação de mutantes não ser automática; e que a maior parte do tempo dedicado ao estudo de caso foi gasto no desenvolvimento de mutantes e execução de seus casos de teste.

Bertolini & Mota (2010) descrevem casos de uso utilizando a *Controlled Natural Language* (CNL), um subconjunto da língua inglesa que permite realizar a descrição quase de forma natural. Esta descrição serve de entrada para a ferramenta TarGeT, criada pelos autores em trabalhos anteriores, que irá gerar os casos de teste em CNL. Então, os casos de teste serão traduzidos em *scripts* de teste para a linguagem alvo (parametrizável). Finalmente, os *scripts* de teste gerados são ajustados para poderem ser executados (bibliotecas de código são importadas e o código de inicialização e término do *script* é incorporado).

Na ferramenta TarGeT, o motor CNL checa algumas inconsistências, gerando um relatório de erros. Se uma palavra não for encontrada no dicionário da CNL e ela se referir ao componente da interface com o usuário, a linguagem de *script* usada deve suportar o componente. Ou seja, há uma relação de dependência entre o dicionário usado e a linguagem de *script* (ou *framework*) alvo.

Como o principal alvo do trabalho é o teste de aplicações móveis para celulares, a seguinte sintaxe é definida:

```
sentence ::= verb nounPhrase
nounPhrase ::= preposition article modifier noun modifier
| article modifier noun modifier
| modifier noun modifier
| modifier noun
| noun modifier
| noun
verb ::= send | call | select ...
article ::= the | a | an
preposition ::= from | to ...
modifier ::= at least | phonebook ...
noun ::= phone | application ...
```

Os oráculos não são gerados automaticamente pela ferramenta e os valores usados nos testes são fornecidos manualmente, para cada componente de interface identificado. Se nenhum valor for fornecido, valores aleatórios são gerados.

O trabalho não aprofunda a análise dos resultados obtidos. Como objetivos

futuros estão o estudo do uso de diferentes algoritmos para a geração de testes e seu impacto no número de defeitos encontrados e no tempo para encontra-los, e a redução de suíte de testes gerados sem perda da eficiência da detecção de defeitos.

Jiang & Ding (2011) descrevem casos de uso utilizando o modelo proposto por Cockburn (2000) e dela é criada uma Tabela de Atividades que servirá como entrada para a construção de uma Máquina de Estados Estendida (MEE) de onde serão derivados os casos de teste.

Na descrição de casos de uso, as sentenças do Fluxo Principal e Fluxos Alternativos são descritas conforme as seguintes estruturas, criadas para a língua inglesa:

- *sujeito + verbo + [objeto]*
- *sujeito + verbo1 + to + verbo2 + [objeto2]*
- *sujeito + verbo + objeto1 + to/from + objeto2*
- *sujeito + verbo + objeto + adjetivo*
- *sujeito + verbo + objeto + verbo no presente*
- *sujeito + verbo + objeto + verbo no passado*

Dessa descrição de casos de uso é gerada uma Tabela de Atividades, que extrai a estrutura das sentenças como uma árvore sintática, usando o Stanford Parser,[18] para um arquivo texto. Somente as principais palavras das sentenças fazem parte da estrutura gerada, que é usada para a geração da MEE. Das sentenças são extraídos os seguintes dados, para formar a MEE:

- Estados iniciais;
- Interações de entrada e de saída;
- Conjunto de variáveis; e
- Conjunto de transições, contendo:
    - Estado origem;
    - Estado destino;

[18] Disponível em: http://nlp.stanford.edu/

- Predicado; e
- Bloco (interação de saída ou *null*, se não houver).

Usando o modelo da MEE gerado, é possível derivar casos de teste. As abordagens geralmente usadas para geração de casos de teste são a Cobertura de Estado, a Cobertura de Transição e a Cobertura de Caminho. No trabalho foi escolhida a Cobertura de Transição, criando-se um algoritmo de geração baseado nesta abordagem.

O trabalho aplica os casos de teste em um sistema de informações sobre o mercado financeiro. Esta aplicação, porém, não é realizada de forma automatizada e um operador deve simular as ações sobre o sistema. Aparentemente, este estudo de caso serviu para os autores avaliarem a aplicação dos casos de teste gerados em um sistema real. Porém, não foi feito um estudo sobre os resultados obtidos. A eficácia dos casos de teste, seu tempo de geração e sua cobertura não foi medida. Há somente uma indicação, na conclusão, que é pretendido melhorar o algoritmo de geração dos casos de teste, minimizando o número de casos e considerando dependências entre requisitos.

Pessoa (2011) constrói uma ferramenta para a documentação de requisitos funcionais em forma de casos de uso e, a partir destes, são gerados testes funcionais para uso com os *frameworks Ruby on Rails*[19] e *Selenium*,[20] visando verificar a correspondência de aplicações *web* com sua especificação.

O preenchimento dos casos de uso é realizado através de uma interface *web* e o formulário contém os mesmos campos citados na seção 2.1. De seus campos, somente o Fluxo Principal e o Fluxo Alternativo são usados na geração dos testes.

É realizada uma classificação quanto ao passo de um fluxo, que pode ser de:

- *Ação*: Representa uma interação do ator com o sistema;
- *Verificação*: Permite checar o comportamento do sistema;
- *Comentário*: Texto usado para enriquecer a documentação, mas sem uso para fins de teste.

[19] Disponível em: http://rubyonrails.org/
[20] Disponível em: http://seleniumhq.org/

A interface com o usuário auxilia o processo de entrada dos passos, exibindo somente as opções aplicáveis a cada tipo de passo.

A geração de cenários é realizada de forma simples, sendo gerado um cenário por fluxo criado. Não há combinação entre fluxos e chamadas a casos de uso (externos) não são tratadas. Ela também está atrelada à geração de casos de teste, sendo feita no mesmo processo.

A geração dos casos de teste é realizada diretamente para *Ruby on Rails* e *Selenium* e há necessidade de informar os valores que serão usados nos testes, antes de sua geração.

Seu gerador e executor de *scripts* de teste realizam os seguintes passos:

1. Iniciar o servidor de testes (*Selenium*);
2. Carregar todos os projetos ativos;
3. Para cada projeto ativo, carregar os casos de uso;
4. Para cada caso de uso, iniciar o navegador;
5. Para cada fluxo do caso de uso, iniciar uma sessão do navegador e carregar seus passos;
6. Para cada passo, montar o *script* de teste (*Selenium*);
7. Executar o *script*;
8. Atualizar status de execução no banco de dados;

A ferramenta permite também agendar a execução dos testes e acompanhar seu progresso usando três visões distintas:

a. *Visão geral*: Onde, para cada projeto de *software*, são mostrados sua quantidade de casos de uso, sua quantidade de fluxos, seu número de fluxos válidos (isto é, que executaram corretamente), seu número de fluxos inválidos e a data e hora da última execução dos testes;

b. *Visão do caso de uso*: Possui basicamente os mesmos dados da visão geral, mas listada por caso de uso. Para cada caso de uso também há uma indicação se este já foi executado.

c. *Visão por passos*: Exibe os passos do caso de uso indicando qualquer passo cuja execução falhou.

Estas visões auxiliam no rastreamento de possíveis falhas que venham a ocorrer na execução dos testes gerados.

A Tabela 6, a seguir, apresenta um **panorama sobre os principais trabalhos encontrados** que construíram uma ferramenta para a geração de casos de teste a partir de casos de uso.

.

Tabela 6 - Panorama das principais ferramentas encontradas

#	QUESTÃO	(PESSOA, 2011)	(JIANG & DING, 2011)	(BERTOLINI & MOTA, 2010)	(GUTIÉRREZ et al., 2008)	(KASSEL, 2006)
1	Usa *somente* casos de uso como fonte para a geração dos testes?	Sim	Sim	Sim	Sim	Sim
2	Qual a *forma de documentação* dos casos de uso?	Português Restrito Semiestruturado	Inglês Restrito e Semiestruturado	Inglês Restrito e Semiestruturado	Inglês Restrito e Semiestruturado	Declarações em *Use Case Mark-up Language*
3	Controla a *declaração* de Casos de Uso?	Sim	Não	Não	Não	Não
4	Dispensa a declaração de fluxos alternativos que avaliam regras de negócio?	Não	Não	Não	Não	Não
5	Quais campos são usados na geração dos testes?	Fluxo Principal, Fluxos Alternativos	Fluxo Principal, Fluxos Alternativos, Pré-condições, Pós-condições	Fluxo Principal, Fluxos Alternativos	Fluxo Principal, Fluxos Alternativos, Fluxos de Erro	Fluxo Principal, Fluxos Alternativos, Pré-condições, Pós-condições, Criticidade e Frequência de Uso
6	Gera cenários automaticamente?	Sim	Sim	Sim	Sim	Sim
7	Gera um cenário para cada fluxo?	Sim	Sim	Sim	Sim	Sim
8	Gera mais cenários, que verifiquem regras de negócio para um mesmo fluxo?	Não	Sim	Não	Não	Sim

#	QUESTÃO	(PESSOA, 2011)	(JIANG & DING, 2011)	(BERTOLINI & MOTA, 2010)	(GUTIÉRREZ et al., 2008)	(KASSEL, 2006)
9	Gera cenários que combinam fluxos?	Não	Sim	Sim	Sim	Sim
10	Gera cenários que incluem mais de um caso de uso?	Não	Não	Não	Não	Não
11	Cenários recebem um peso, para priorização?	Não	Não	Não	Não	Sim
12	Há métricas de cobertura para os cenários?	Não	Não	Não	Sim	Sim
13	Gera casos de teste semânticos?	Não	Sim	Não	Não	Sim
14	Gera valores para os casos de teste automaticamente?	Não	Não	Não	Não	Não
15	Há métricas de cobertura para os casos de teste?	Não	Não	Não	Não	Sim
16	Casos de teste são gerados num formato independente de linguagem ou *framework*?	Não	Sim	Não	Não	Sim

#	QUESTÃO	(PESSOA, 2011)	(JIANG & DING, 2011)	(BERTOLINI & MOTA, 2010)	(GUTIÉRREZ et al., 2008)	(KASSEL, 2006)
17	Gera código de teste?	Sim	Não	Sim	Sim	Não
18	O código de teste está disponível para quais linguagens ou *frameworks*?	Ruby on Rails com Selenium.	-	BadBoy[21]	Java com JWebUnit[22]	-
19	Há uso de métricas sobre o código gerado?	Não	-	Não	Não	-
20	Os resultados da execução do código gerado são rastreados?	Sim	-	Não	Não	-
21	Há análise de desempenho?	Não	Não	Sim	Não	Sim
22	Há análise de esforço da geração de testes, comparando-o a outros meios (ex.: *capture & replay*)?	Sim	Não	Não	Não	Não

[21] Disponível em: http://www.badboy.com.au/
[22] Disponível em: http://jwebunit.sourceforge.com

Pode-se concluir, de forma geral, que há diversos aspectos a serem explorados e evoluídos, havendo campo para pesquisas e criação de novas ferramentas que os coloquem em prática. Portanto, o presente trabalho buscou construir uma solução que pudesse preencher algumas das lacunas existentes.

3.3.Comparativo

A solução construída verificou problemas, soluções e considerou diversas ideias (mesmo não implementadas) de outros trabalhos e procurou reutilizá-las ou adaptá-las para a confecção da ferramenta. As principais influências, diferenças, contribuições e restrições são analisadas a seguir.

3.3.1.Principais influências

Este trabalho foi influenciado por diversos outros, dos quais se pode recordar:

- A descrição textual dos casos de uso foi inspirada no layout de formulário proposto por Staa (2011), adaptado de Cockburn (2000);
- A geração de cenários e de casos de teste foi inspirada no trabalho de Heumann (2001), assim como na maioria dos trabalhos analisados, apesar de seu trabalho não descrever precisamente como realizá-la de forma automatizada (apenas informalmente);
- Kassel (2006) inspirou o uso da Criticidade, da Frequência de Uso, do estabelecimento da Importância; a adoção de máquinas de estado para Pré-condições e Pós-condições dos casos de uso; e inspirou a organização dos fluxos;
- Caldeira (2010) e Pessoa (2011) inspiraram a estrutura dos passos dos casos de uso. O trabalho de Pessoa (2011) também inspirou o processo de coleta de resultados da execução dos testes.

3.3.2.Principais diferenças

Entre as principais diferenças da solução proposta em relação aos trabalhos descritos na seção 3.2.

1. Apresentação de um processo bem definido, completo e totalmente automatizado;
2. Descrição das regras de negócio na especificação dos casos de uso, permitindo:
    a. Gerar casos de teste que visam verificá-las;
    b. Gerar valores (válidos e inválidos), que visam explora-las;
    c. Não representar fluxos alternativos que realizam verificações de regras de negócio, uma vez que é possível verificá-las pelos testes gerados a partir delas.
3. Uso de fontes de dados externas na composição das regras de negócio, permitindo:
    a. A simulação de condições reais de uso, com dados contendo a mesma estrutura dos utilizados em produção;
    b. A simulação de condições que dependem de dados presentes no sistema;
    c. A variação dos dados utilizados nos testes, aumentando as chances de encontrar defeitos.
4. Geração de cenários que envolvem repetições de fluxos (*loops*), com o número de repetições parametrizável (com *default* em uma repetição):
    a. Gutiérrez *et al.* (2008) leva em conta *loops*, mas não indica a parametrização do número de repetições. Esta parametrização é importante, pois influencia diretamente o número de cenários gerados e, consequentemente, seu tempo de geração.
5. Geração de cenários que envolvem mais de um caso de uso. Nenhum dos trabalhos investigados considerou a combinação de cenários envolvendo diferentes casos de uso;
6. Geração de testes semânticos com nomes correlatos ao tipo de verificação a ser realizada, permitindo:
    a. Que o desenvolvedor entenda o que o teste verifica, pelo seu nome;

   b. Que a possível manutenção dos testes seja mais fácil, pela facilidade de rastreamento disponibilizada pelo esquema de nomeação adotado.

7. Uso de vocabulário configurável, permitindo adaptá-lo ao *framework* de testes desejado;
8. Uso de perfis, que possibilitam traduzir o vocabulário esperado pelo *framework* de testes, permitindo que a documentação dos casos de uso seja feito no idioma desejado.

3.3.3.Principais restrições

As principais restrições da solução construída são:

1. A solução não simula os Fluxos de Exceção Recuperável (FER) e os Fluxos de Exceção Irrecuperável (FEI), detalhados no Capítulo 4, pelo fato destes tipos de exceção serem de difícil (senão impossível) simulação através de testes funcionais;
2. As regras de negócio não aceitam expressões que envolvam cálculos, fórmulas matemáticas ou uso de condicionais (ex.: *if-then-else*);
3. A abrangência dos tipos de interface gráfica passíveis de teste pela ferramenta é proporcional aos *frameworks* de testes de interface utilizados. Assim, é possível que determinados tipos de interface com o usuário, como as criadas para *games* ou aplicações multimídia, possam não ser testadas por completo, se o *framework* de testes escolhido não suportá-los, ou não suportar determinadas operações necessárias para seu teste.