3 Revisão bibliográfica

São muitos os trabalhos existentes sobre testes baseados em modelo, para uma variedade deles, assim como sobre o uso do critério Análise de Mutantes (AM) para verificar e complementar os casos de teste já selecionados. Alguns destes trabalhos estão mais voltados para a teoria e outros aplicam o estudado como prova de conceito em alguma ferramenta.

Na realidade, cada classe de modelo apresenta deficiências e pontos fortes, o que depende do tipo de problema e ambiente ao qual será aplicada. O ideal é que se descubra quais são os modelos mais apropriados para quais tipos de problemas a resolver.

Para algumas linguagens, como Java, e plataformas como a Web, as ferramentas voltadas para testes de interface estão bastante evoluídas, sendo utilizadas em ambientes de desenvolvimento de softwares de grande escala. No entanto, em linguagens como C++ esse ferramental ainda é escasso, principalmente se considerados os diversos tipos de toolkits de interface gráfica disponíveis para ela.

PUC-Rio - Certificação Digital Nº 1212388/CA

3.1 Teste de GUI a partir de modelo

Em [15] foi realizado um estudo sobre a eficácia de casos de teste para GUI, na detecção de falhas em softwares de rápida evolução. Para isso focou o estudo nos seguintes aspectos sobre o teste automático em GUI: Como o tamanho de uma suíte de teste impacta na eficácia da detecção de falhas; Como a complexidade de um oráculo influencia na eficácia; E que características de falhas podem e não podem ser detectadas em testes de GUI.

A geração do modelo é automática e realizada pelo framework DART, que analisa a estrutura da GUI e extrai os elementos e suas propriedades. Internamente a representação da GUI é um conjunto de triplas <nome, propriedade, valor> e grafos interação-evento. O framework computa o número total de possíveis casos de teste que podem ser executados. Como esse número é muito elevado, o testador especifica quantos devem ser de fato gerados. No entanto ele não tem acesso a quais casos de teste descartar, apenas define o número a que deve ser reduzido. Alguns pontos negativos sobre quando o modelo é gerado automaticamente, são: 1) Não se tem uma visão da estrutura da GUI e das possíveis interações; 2) Ao automatizar cada vez mais etapas, principalmente as de nível mais alto, os custos de

refinamentos manuais a serem realizados posteriormente buscando o aumento da acurácia e as chances de não se atingir a completude dos casos gerados, tendem a aumentar.

Outra característica do framework que gera o modelo é que, para gerar automaticamente o modelo, ele interpreta arquivos com toolkit de interface em Java especificamente.

Outras considerações feitas no artigo são que os resultados apresentados são válidos para aplicações onde o código é intensamente voltado para a GUI (aprox. 80%) e concluem que estes resultados deverão ser diferentes para aplicações que apresentam camada lógica complexa e GUI relativamente simples. Outro ponto levantado é a necessidade de serem feitos ajustes manuais nos casos de teste gerados, devido ao alto nível de detalhe do código necessário para executar corretamente os fluxos de uma aplicação. Isso é uma característica observada na maioria dos trabalhos em que se têm casos de teste gerados automaticamente.

Por fim, o ponto que mais afasta o trabalho em [15] ao aqui realizado é o fato de que no primeiro, o modelo extraído automaticamente apenas considera e implementa eventos intra-componente, não modelando eventos de interação com outras funcionalidades do sistema. E relata que também não está em seu escopo, lidar com eventos de natureza não determinística.

Em [16] a geração dos casos de teste é baseada em diagramas de Casos de Uso e de Atividades da UML (Unified Modeling Language). Os diagramas de casos de uso descrevem o relacionamento ente os diversos casos de uso especificados e os atores que interagem com a aplicação. Os diagramas de atividades são usados para capturar a lógica de um único caso de uso. O conjunto de diagramas de atividade representa o comportamento geral especificado para a aplicação e é a base para testar as diferentes funcionalidades e regras de negócio descritas na especificação dos casos de uso.

A combinação dos dois diagramas tem o potencial de gerar um grande número de casos de teste. O artigo apresenta duas formas de gerenciar o número de testes. Na primeira os dados de entrada são escolhidos segundo um critério chamado *Category-Partition,* o que implica diretamente na geração de caminhos possíveis. Na segunda consiste em estarem disponíveis diferentes configurações para cobertura do grafo.

PUC-Rio - Certificação Digital Nº 1212388/CA

Para finalizar o processo, as suítes de teste geradas são executadas por uma ferramenta própria sobre o SUT. Antes da geração dos casos de teste, um critério de cobertura para o modelo é escolhido, bem como o critério para a geração dos dados de entrada. Os critérios que provém uma cobertura mais fraca e que geram menos casos de teste são escolhidos a fim de se obter um número razoável de casos de teste que não impliquem em maiores custos.

Uma desvantagem identificada neste trabalho é que apesar de haver um modelo visual da funcionalidade, ele não pode ser simulado. É um modelo estático. Com isso, não se pode obter uma visualização prévia do modelo, não sendo possível validá-lo, nem verificar a especificação.

O tipo de modelo mais comumente utilizado para testes de software são as máquinas de estado. Elas modelam o comportamento em termos de seus estados abstratos ou concretos e são representados tipicamente como diagramas estado transição.

Em [5] os autores desenvolveram um tipo de máquina de estado finita que diminui o número de estados abstratos adicionando variáveis ao modelo (tal como em uma Rede de Petri de alto-nível). No entanto, eles consideram que as FSMs por vezes utilizadas apresentam problemas de extensão quando trabalhando com GUIs muito grandes.

Em [17] foi apresentado um processo de geração de casos de teste baseado em tabelas de decisão. O processo inicia com o editor gráfico da tabela, passando pelo gerador automático dos scripts de teste e culminando com uma ferramenta capaz de executar os scripts no software a ser testado. A área do software ao qual se direciona este trabalho é também a interface e os scripts gerados são para a linguagem Java.

As dificuldades observadas neste trabalho ligadas ao uso da tabela de decisão incluem: A dificuldade em avaliar alguns detalhes do comportamento do sistema que não ficam claros através da especificação do caso de uso, mas que são bastante importantes para a construção da tabela de decisão. E a cobertura da suíte gerada foi bem próxima aos testes manuais usados na avaliação. O motivo disso pode estar relacionado ao fato de a geração dos casos de teste ser manual e não ser possível obter a partir da tabela de decisão, um comportamento cíclico da interface, dando possibilidade de casos de teste mais longos serem gerados.

PUC-Rio - Certificação Digital Nº 1212388/CA

O assunto talvez mais estudado sobre geração de casos de teste para GUI está relacionado a critérios de cobertura, os quais incluem: Todos os eventos, todas as interações entre eventos, todos os estados, todos os caminhos, dentre outros. A ideia é produzir casos de teste que cubram toda a rede modelada esperando-se cobrir todas as possíveis interações com a GUI, cercando portanto, todas as possibilidades de erros. Esta abordagem no entanto, tem uma óbvia limitação prática e deve ser sempre seguida por um método de filtragem dos casos de teste, antes ou depois da geração. Além disso, existe a dúvida sobre a eficácia dos casos de teste gerados, pois o estado de uma GUI após uma transição pode ser diferente dependendo da combinação de eventos executados anteriormente (ou, mais explicitamente, do estado logo anterior). Muitos estudos ainda estão em andamento e nenhum é conclusivo sobre a cobertura que gere os casos de teste mais eficazes.

3.2 GUI como Rede de Petri

Apenas um trabalho que menciona a modelagem de interfaces gráficas como Rede de Petri foi encontrado. Em [18] é apresentado um critério de cobertura para testes em GUI, baseado no modelo Rede de Petri do tipo Predicado/Transição. O trabalho expande a abordagem baseada em evento (event-based) descrita em [19] e adiciona outro critério de cobertura chamado all-states, que considera também o estado da rede antes e após um disparo de transição. No entanto, o trabalho não define como a rede pode ser construída, de forma a representar a GUI de maneira eficiente, assim como não realizou experimentos práticos. Por isso não demonstrou a viabilidade e a eficácia da abordagem de cobertura híbrida utilizada.

3.3 Ferramentas de teste em GUI

Ferramentas semiautomáticas de teste de unidade tais como JFCUnit, Abbot e Pounder são usadas para criar manualmente testes de GUI, os quais são depois executados automaticamente. Assertivas são inseridas nos casos de teste para determinar se as classes e métodos na unidade sob teste funcionam corretamente [20].

Um tipo de ferramenta bastante difundida no contexto de testes em GUI são as chamadas *capture/replay*, onde o testador ou um usuário executa os fluxos desejados na GUI, um por um, de forma transparente. A ferramenta grava as ações

PUC-Rio - Certificação Digital Nº 1212388/CA

e o estado da interface, onde cada sequência de eventos é um caso de teste e gera os scripts de teste de forma automática. Usando uma ferramenta de teste automatizado associada à ferramenta de gravação, pode-se posteriormente executá-los de forma automática. A criação de teste é manual e, a cobertura e o espaço de estados da GUI explorados por esses casos de teste, dependem muito da experiência e conhecimento dos testadores e da qualidade das sessões de usuário. Testadores que utilizam estas ferramentas apresentam normalmente um pequeno número de testes criados, sendo esses também curtos.

Embora a reprodução seja automatizada, existe um esforço envolvido na criação de scripts de teste e na edição dos testes para fazê-los funcionar quando o software evolui.

Algumas ferramentas *capture/replay* salvam a coordenada do mouse onde o evento foi disparado (*widget*). Considerar uma ferramenta deste tipo com esta característica deve ser feito com atenção, porque qualquer mudança de layout da interface irá invalidar o teste, exigindo sua manutenção. Já os tipos que guardam os eventos pelo nome do *widget* ou por reconhecimento de padrões, baseando-se em *screenshots,* são mais seguros dado que a posição dos elementos na tela torna-se indiferente.

A ferramenta Sikuli apresentada em [21] permite o uso de *capture/replay* ou a construção de scripts manualmente para interface gráfica. O diferencial apresentado é a possibilidade de se distinguir os elementos de interface através de *screenshots*, aliado ao algoritmo de reconhecimento de padrões dito eficaz para variações de S.O., temas e resoluções. Como não baseia-se em elementos do código, serve para programas em qualquer linguagem. No entanto, ainda apresenta problemas no uso em softwares reais, em relação as *delays* existentes na hora em que a verificação vai ser realizada, quando ocorrem processamentos que podem levar tempos variados.

3.4 Análise de mutantes no nível funcional e de sistema.

Tradicionalmente a análise de mutantes é aplicada ao nível de unidade (funções e métodos), onde os operadores geradores de mutações representam erros cometidos por programadores numa unidade de software. Há ainda as abordagens de mutação que são voltadas para o nível de integração e para o nível de classe. Na mutação no nível de integração, os operadores são definidos de forma a re-

PUC-Rio - Certificação Digital Nº 1212388/CA

presentar os erros cometidos na conexão entre unidades do programa. Já a mutação no nível de classe foi alvo de pesquisas que tentavam criar operadores voltados para características típicas de programas orientados a objetos, tais como polimorfismo e herança. No entanto, pouco estudo foi realizado sobre a eficácia e limitações da mutação no nível de classe na prática. [22]

Em [22] apresenta-se uma experiência usando o teste de mutação para medir a eficácia de um gerador de dados de teste automatizado para a análise de modelos de "features". Os experimentos foram conduzidos usando o FaMa, uma ferramenta *opensource* em Java que analisa esses modelos. Três classes foram totalmente mutadas usando mutação tradicional e no nível de classe. Os resultados foram avaliados contrastando cada operador utilizado com o seu *score* de mutação e com a percentagem de mutantes equivalentes gerados para cada operador.

Um resultado importante levantado pelo artigo e que deve ser explicitado é que a porcentagem de mutantes equivalentes gerados pelos operadores de nível de classe (45,4%) é muito maior do que o gerado com operadores tradicionais (13,4%). Este resultado é também maior que as percentagens reportadas em estudos semelhantes, sugerindo que entre 5% e 15% de mutantes gerados são equivalentes. Isso faz com que operadores de mutação de nível de classe sejam menos atraentes para a experimentação, uma vez que geram uma porcentagem maior de mutantes equivalentes e menos mutantes do que os operadores tradicionais, reduzindo a parcela total de mutantes úteis [22].

Segundo o estudo em [23] mais de 86% dos mutantes de classe foram equivalentes, sendo um total de aproximadamente 49.000 mutantes de seis softwares avaliados. Este resultado é ainda mais forte e indicativo de que operadores para nível de classe podem não ser uma boa estratégia.

Em [24] foi introduzido o conceito de mutação no nível funcional e de sistema, onde o objetivo é a detecção de falhas no nível funcional, ou seja, falhas que não estão presentes em níveis mais baixos de abstração. Para isso, definem operadores para o nível do sistema, dentre os quais pode-se citar: Operadores relacionados a configuração, interação entre classes e GUIs.

Neste trabalho a ideia foi aplicar a análise de mutantes para validar os casos de teste gerados a partir do modelo Rede de Petri representando uma interface gráfica. Como a natureza destes testes é de nível funcional, abrangendo interações entre classes e código específico da lógica de interface, utilizamos as mutações de

PUC-Rio - Certificação Digital Nº 1212388/CA

nível funcional e também de nível de unidade. Os operadores de nível de unidade foram coletados da literatura atual. Os operadores de nível funcional foram definidos de acordo com as características de implementação do software sob teste.

Alguns trabalhos, como [25] e [26] utilizam outra abordagem e aplicam a análise de mutantes na Rede de Petri, definindo operadores de mutação para a rede e gerando redes mutantes. O objetivo destes trabalhos é validar a especificação.

No caso deste trabalho, partimos do pressuposto que a especificação está correta e que a rede está corretamente modelada. Através da simulação, capturando as ações e os estados, obtemos os casos de teste. Observamos se num primeiro momento já consegue identificar algum defeito e aplicamos então a AM no código do software sendo testado para verificarmos o quão suficiente/adequada é a suíte de teste gerada.

A abordagem estudada e aplicada neste trabalho visa contornar as deficiências apresentadas neste capítulo.

O problema da explosão do número de casos de teste gerados por modelos que geram casos de teste baseado em algum critério de cobertura, exigindo um grande esforço no ajuste dos scripts gerados e na execução posterior dos testes, é contornado no nosso método, pela geração de casos de teste mais longos e em menor número. Além disso, a geração é baseada na abordagem aleatória, técnica bastante defendida em alguns trabalhos, como em [27] e [28].

O ajuste manual dos scripts gerados ainda é necessário neste trabalho, custando cerca de uma hora. Mas com um esforço pequeno de evolução na ferramenta que gera os scripts, pode-se diminuir esse tempo para poucos minutos.

Quando as funcionalidade de GUI têm os seus requisitos alterados, o custo para gerar novamente os testes é baixo, sendo necessário apenas a atualização no modelo. Mesmo o modelo sendo muitas vezes grande, a atualização nele é mais prática e transparente do que no código do script, ainda mais quando os scripts são grandes e com muitos casos de teste, que nesse caso torna-se um trabalho de atualização muito custoso.

PUC-Rio - Certificação Digital Nº 1212388/CA