UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

RODRIGO FUNABASHI JORGE

Estudo, Definição e Proposta de Representação de Interface Web Visando à Atividade de Teste de Software

Goiânia
2016

sistema de bibliotecas ufg

UFG

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR AS TESES E DISSERTAÇÕES ELETRÔNICAS (TEDE) NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

1. Identificação do material bibliográfico: **[ ] Dissertação** **[X] Tese**

2. Identificação da Tese ou Dissertação

Autor (a):	Rodrigo Funabashi Jorge						
E-mail:	rodrigofunabashijorge@gmail.com						
Seu e-mail pode ser disponibilizado na página? [X]Sim [ ] Não							
Vínculo empregatício do autor	Universidade Federal de Mato Grosso do Sul						
Agência de fomento:	Fundação de apoio ao desenvolvimento do ensino, ciência e tecnologia do estado de mato grosso do sul				Sigla:	FUNDECT	
País:	Brasil	UF:	MS	CNPJ:	02.776.669/0001-03		
Título:	Estudo, Definição e Proposta de Representação de Interface Web Visando à Atividade de Teste de Software						
Palavras-chave:	Teste Baseado em Modelos, Teste GUI, Geração de Dados de Teste e Automatização da Atividade de Teste						
Título em outra língua:	Study, Definition and Proposal of Web Interface Representation Aiming at the Software Testing Activity						
Palavras-chave em outra língua:	Model Based Testing, GUI Test, Test Data Generation and Automation Test Activity						
Área de concentração:	Ciência da Computação						
Data defesa: (dd/mm/aaaa)	01/04/2016						
Programa de Pós-Graduação:	Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás						
Orientador (a):	Prof. Dr. Auri Marcelo Rizzo Vincenzi						
E-mail:	auri@dc.ufscar.br						

3. Informações de acesso ao documento:

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF ou DOC da tese ou dissertação.

O sistema da Biblioteca Digital de Teses e Dissertações garante aos autores, que os arquivos contendo eletronicamente as teses e ou dissertações, antes de sua disponibilização, receberão procedimentos de segurança, criptografia (para não permitir cópia e extração de conteúdo, permitindo apenas impressão fraca) usando o padrão do Acrobat.

Data: 20/05/2016

Assinatura do (a) autor (a)

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

RODRIGO FUNABASHI JORGE

Estudo, Definição e Proposta de Representação de Interface Web Visando à Atividade de Teste de Software

Tese apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Doutor em Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Auri Marcelo Rizzo Vincenzi

Goiânia
2016

Ficha catalográfica elaborada automaticamente
com os dados fornecidos pelo(a) autor(a), sob orientação do Sibi/UFG.

Funabashi Jorge, Rodrigo
Estudo, Definição e Proposta de Representação de Interface Web Visando à Atividade de Teste de Software [manuscrito] / Rodrigo Funabashi Jorge. - 2016.
CXLV, 145 f.: il.

Orientador: Prof. Dr. Auri Marcelo Rizzo Vincenzi.
Tese (Doutorado) - Universidade Federal de Goiás, Instituto de Informática (INF) , Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2016.
Bibliografia.
Inclui algoritmos, lista de figuras, lista de tabelas.

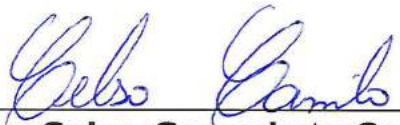
1. Teste Baseado em Modelos. 2. Teste GUI. 3. Geração de Dados de Teste. 4. Automatização da Atividade de Teste. I. Marcelo Rizzo Vincenzi, Auri, orient. II. Título.

Rodrigo Funabashi Jorge

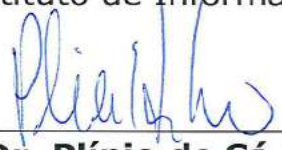
Estudo, Definição e Proposta de Representação de Interface Web Visando a Atividade de Teste de Software

Tese defendida no Programa de Pós-Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Doutor em Ciência da Computação, aprovada em 01 de abril de 2016, pela Banca Examinadora constituída pelos professores:

Prof. Dr. Auri Marcelo Rizzo Vincenzi
Instituto de Informática - UFG
Presidente da Banca

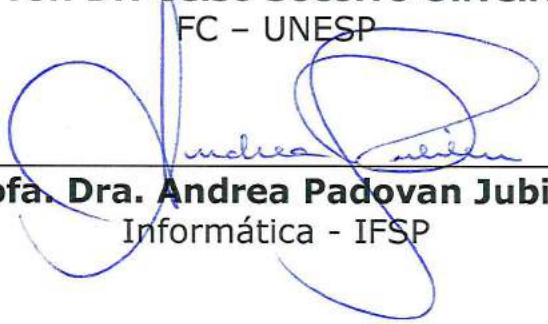


Prof. Dr. Celso Gonçalves Camilo Júnior
Instituto de Informática – UFG



Prof. Dr. Plínio de Sá Leitão Júnior
Instituto de Informática – UFG

Prof. Dr. Celso Socorro Oliveira
FC – UNESP



Profa. Dra. Andrea Padovan Jubileu
Informática - IFSP

Rodrigo Funabashi Jorge

Possui mestrado em computação pelo Instituto de Ciências Matemáticas e de Computação (ICMC) da Universidade de São Paulo (USP). Hoje é professor adjunto da Universidade Federal de Mato Grosso do Sul (UFMS), tendo Engenharia de Software como área de pesquisa.

Ao meu filho, Pedro Paulo Pastore Jorge, pelo amor incondicional e que me traz força e razão para tentar ser uma pessoa cada vez melhor, seja como pai ou como profissional.

Para minha esposa, Camila Dib Silva Jorge, por ser minha felicidade de todos os dias. Minha principal incentivadora e quem me deu forças para superar os obstáculos de um doutorado, fazendo com que não desanimasse nos momentos difíceis.

Agradecimentos

Deus, por sempre ter me agraciado e acompanhado em tudo que sempre pensei em ser e conquistar. Por mostrar a força do seu amor a ponto de ter enviado seu único filho para morrer por nós. Por ser um Pai maravilhoso e me concedesse o dom da vida, permitindo que eu chegasse até aqui. Por isso, "Deem graças ao Senhor porque ele é bom; o seu amor dura para sempre" (Salmos 107:1).

Ao meu filho Pedro Paulo, por compreender minha ausência em alguns momentos. Desculpe-me, filho, se em algumas situações, estive impaciente ou nervoso.

A minha esposa, Camila, pelo amor, dedicação, cumplicidade, incentivo e compreensão em todos os momentos, principalmente nos mais difíceis, que foram muitos durante todo o processo.

Aos meus pais, Elcio e Sônia, pelo amor incondicional, por investirem e acreditarem em mim e por serem meus exemplos de vida. Sem vocês jamais teria chegado aqui.

A minha irmã Patrícia, que sempre me incentivou e torceu por mim profissionalmente. Além de tudo, durante esse período de estudos, presenteou-me com a minha linda afilhada Olívia.

A minha sogra Jamile, que está sempre preocupada comigo, tentando me acalmar em momentos que eu não via uma luz no fim do túnel.

Ao meu orientador, professor Dr. Auri Marcelo Rizzo Vincenzi, pela orientação, amizade, compreensão e profissionalismo durante todo o período deste trabalho.

Aos demais professores do Instituto de Informática (INF) da Universidade Federal de Goiás (UFG). Em especial ao professor Dr. Celso Camilo, pela contribuição dada no trabalho nos estudos relacionados à inteligência artificial.

Em especial ao meus amigos Marcos Paulo e Rafael, pela grande amizade e por estarem sempre dispostos a ajudar, mesmo em momentos difíceis. Vamos que vamos, Marcão!!

Ao professor Dr. Marcelo Henriques, coordenador geral do programa de doutorado e professor da Faculdade de Computação (FACOM) da Universidade Federal de Mato Grosso do Sul (UFMS), pela cordialidade e ajuda de sempre.

Em especial aos amigos José Carlos (Zé Galinha), André (Atleta), Honório Jacometto, Ana Raquel (Copetti), Gustavo e Diogo, por sempre me cederem a casa de vocês em minhas viagens para as orientações. Também ao professor Rossine pela ajuda na revisão dessa tese. Você é o cara!

Aos meus amigos dos momentos de estudos, descontrações e desabafos: Ricardo (Koim), Éder (Cabeleira), Cadu, Eron, Igor (cunhado), meu primo Daniel, Neto (primo), Pedro Paulo, Charles, Thiago, Max, Camilo (doutor), Weber, Renato, Gilmarzinho, Cristiano, Hércules, Betão e Liana. Saibam que os momentos que passamos juntos sempre estarão guardados em meu coração.

Aos meus amigos e amados irmãos em Cristo da igreja Santo Antônio e das equipes dos acampamentos. São tantos que não caberiam nesse agradecimento. Sei que todos torcem pelo meu sucesso!

Aos funcionários do INF–UFG e da FACOM–UFMS, pela disposição e atenção de todos que me ajudaram ou torceram por mim de forma direta ou indiretamente.

E a Fundação de Apoio ao Desenvolvimento do Ensino, Ciência e Tecnologia do Estado de Mato Grosso do Sul (FUNDECT), pelo apoio financeiro.

Sonhos determinam o que você quer. Ação determina o que você conquista,
Aldo Novak.

Resumo

Funabashi Jorge, Rodrigo. **Estudo, Definição e Proposta de Representação de Interface Web Visando à Atividade de Teste de Software**. Goiânia, 2016. 143p. Tese de Doutorado. Instituto de Informática, Universidade Federal de Goiás.

O objetivo principal da Engenharia de Software é dar subsídios para desenvolvimento de software, desde a sua especificação até sua implantação e manutenção, aplicando métodos, processos e ferramentas, buscando uma maior qualidade no software produzido. Uma das atividades para se buscar a qualidade desejada é a atividade de teste de software. Essa atividade pode se tornar bastante complexa, dependendo das características e dimensões do produto de software a ser desenvolvido e, desse modo, está sujeita a diversos tipos de problemas que acabam resultando na obtenção de um produto com defeitos, prejudicando assim a qualidade do mesmo. Apesar da complexidade e das limitações existentes, encontram-se na literatura diferentes técnicas que podem ser utilizadas para gerar dados de teste para satisfazer os diversos critérios de teste de software existentes, procurando assim reduzir o custo dos testes. Porém, a geração de dados de teste é um problema indecidível, devido à complexidade e o tamanho de programas. Um dos fatores que aumentam a complexidade é o uso de interfaces do usuário (UI – *User Interfaces*), presentes em muitas aplicações. Essa complexidade é resultante do elevado número de combinações de entrada disponível, tornando praticamente impossível realizar os testes UI de forma manual.

Dentre as alternativas que viabilizam a automatização uma das mais reconhecidas e vantajosas é o teste de UI baseado em modelos. Esta técnica passa pela construção de um modelo a partir da estrutura da interface da aplicação a ser testada e os dados de teste são gerados a partir desse modelo. Porém, um fator problemático desta abordagem reside na construção do modelo. Este processo pode ser demorado e dispendioso e, em alguns casos, pode ser bastante complicado e não atingir um objetivo satisfatório por não conseguirem representar, por meio do modelo proposto, características reais da aplicação. Ao estudar o estado da arte de teste UI, observou-se que existem ferramentas que permitem realizar tais testes automaticamente, mas essas ainda possuem algumas limitações, principalmente decorrentes do modelo de representação da interface adotado por elas. Desse modo, a proposta dessa tese é propor um modelo de representação de UI que traga benefícios em relação às representações hoje existentes na literatura. Com a proposta deste modelo é possível representar com o maior nível de detalhes uma interface gráfica para aplicações de software. Um estudo preliminar, comparando o modelo proposto com outros disponíveis na literatura, evidencia os benefícios alcançados.

Palavras–chave

Teste Baseado em Modelos, Teste GUI, Geração de Dados de Teste e Automatização da Atividade de Teste.

Abstract

Funabashi Jorge, Rodrigo. **Study, Definition and Proposal of Web Interface Representation Aiming at the Software Testing Activity**. Goiânia, 2016. 143p. PhD. Thesis. Instituto de Informática, Universidade Federal de Goiás.

The main purpose of software engineering is to subsidy the software development, from its specification to its implementation and maintenance, by applying methods, processes and tools seeking for a higher quality software product. One of the activities to get the desired quality is software testing. This activity can become very complex, depending on the characteristics and dimensions of the software product under developed and thus, is subjected to various kinds of problems which, eventually, may result on a product with faults, jeopardizing its quality. Despite the complexity and limitations of testing, there are in the literature different techniques that can be used to generate test data to satisfy several testing criteria, aiming at to reduce the cost of testing. However, generation of test data is an undecidable problem due to the complexity, constraints, and size of programs. One of the factors that increase the complexity is the use of user interfaces (UI), present in many applications. This complexity is a result of the high number of available input combinations, making it virtually impossible to hold the UI tests manually.

Among the alternatives that enable the automation of the most recognized and advantageous is the UI based testing models. This technique involves the construction of a model to abstract the UI elements, its interactions, and structure to be tested. From this model, the test data can be generated. However, a troublesome factor in this approach lies in building the model. This process can be costly and time consuming. Additionally, even after the effort, the model can be incomplete and may not represent precisely the actual characteristics of the application. When studying the state of the art for testing UI, we noted that there are tools that allow to perform such testing automatically. But they also have some limitations, mainly arising from the representation model adopted. Thus the purpose of this thesis is to propose a UI representation model that brings benefits over existing representations in today literature, evolving the state of art on this area. With the proposal of this model we can represent, with the greatest level of detail, a graphical interface for web software applications. A preliminary study comparing the model with others available in the literature, highlights the benefits achieved.

Keywords

Model Based Testing, GUI Test, Test Data Generation and Automation Test Activity.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos de Programas

CAPÍTULO 1

Introdução

Este capítulo trata da apresentação do trabalho fazendo uma breve introdução de conceitos no contexto de automatização da atividade de teste de software, focando em testes baseados em UI– *User Interfaces*. Em seguida, são apresentados os objetivos e justificativas que motivaram o foco do trabalho. Também são descritas a metodologia utilizada e a estrutura organizacional da tese.

1.1 Contextualização

A sociedade atual está se tornando mais e mais dependente de sistemas de software. Eles estão presentes em praticamente todas as partes da sociedade moderna, desde aviões e carros até em ações de compras pela Internet. Esta crescente implantação de sistemas de software torna a nossa vida cada dia mais dependente de seu funcionamento sem falhas, sendo isso diretamente relacionado à sua aderência aos requisitos dos clientes. Segundo Pressman e Maxim (2014), os problemas resultantes de uma má interpretação dos requisitos dos clientes são os mais caros para serem corrigidos.

Uma das atividades mais comuns para aumentar a confiança de sistemas de software é a de teste. Essa atividade envolve executar o sistema com um conjunto de entradas e avaliar se o conjunto de saídas é válido, ou seja, se está de acordo com a especificação. Existem diferentes técnicas de testes como o teste de caixa branca (teste estrutural) e o teste de caixa preta (teste funcional). No teste caixa branca, o conhecimento do código-fonte é utilizado para derivar um conjunto de teste que execute o código-fonte perante determinado critério. Nas técnicas de caixa preta, o produto de software é visto como uma caixa fechada que recebe entradas e produz saídas, e o conjunto de testes é derivado da especificação de requisitos e/ou outros artefatos que descrevem as funcionalidades presentes no produto em teste.

A atividade de teste pode ser considerada como uma sequência de três fases. A primeira delas é o teste de unidade, no qual cada unidade do software é testada individualmente, buscando-se evidências de que ela funcione adequadamente. A próxima fase, o teste de integração, é uma atividade sistemática para integrar as unidades componentes da

estrutura do software, visando a identificar defeitos de interação entre elas. Finalizando, o teste de sistema verifica se todos os elementos do sistema combinam-se adequadamente e se a função/desempenho global do mesmo é atingida (PRESSMAN; MAXIM, 2014).

Dentro do contexto de teste de sistema, surge o teste de UI. Apesar da teoria pregar que os testes sejam realizados da menor unidade para ao sistema, na prática, em geral, as empresas concentram esforços nessa última fase.

A interface do usuário é denominada de forma diferente dependendo do tipo de aplicação e da plataforma. No contexto deste trabalho caracterizam-se os seguintes tipos de UI: GUI (*Graphical User Interface*), para programas *desktops*, WUI (*Web User Interface*), para aplicações *web* e CLI (*Command-Line Interface*), para interfaces de linhas e comando, que não será considerado nesse trabalho.

O teste baseado em WUI é o foco deste trabalho e tem como principal objetivo, a partir das possibilidades de eventos da interface, testar determinada aplicação. Porém, as dificuldades para isso são grandes devido, principalmente, à uma interface gráfica ter muitas operações que podem ser usadas como teste. Essas dificuldades motivaram esse trabalho e são apresentadas na seção a seguir.

1.2 Justificativas

Sistemas de software modernos compreendem vários componentes, que interagem uns com os outros para realizarem tarefas. O comportamento correto desses componentes é frequentemente verificado por meio de testes de integração das unidades e do sistema. Muitos dos aplicativos de hoje têm um componente especial na forma de UI. A UI é uma interface que consiste de elementos de controle chamados *widgets*, também chamados de objetos de uma interface, como por exemplo botões, itens de menu e caixas de texto. A UI é frequentemente a única parte do software por meio da qual o usuário interage com a aplicação. Com isso, é necessário testar exaustivamente esta interface, a fim de garantir a qualidade do produto, por meio da criação de dados de teste sob a forma de sequências de entrada. Uma sequência de entrada para uma aplicação UI é uma sequência de ações, como preenchimento de um campo de texto, ou um clique em um botão de controle, uma operação de arrastar e soltar, dentre outras.

A Figura 1.1 ilustra uma sequência de entrada para o Microsoft Word que faz com que o documento aberto seja impresso. Em um cenário típico de testes, os testadores começam pela definição de conjunto de testes inicial que, geralmente, incluem cenários comuns, como impressão de um documento, preencher formulários e alimentar o banco de dados. Esses testes são criados manualmente por *scripts* ou gerados automaticamente por ferramentas de captura/reprodução (*capture/playback*) que permitem, durante a fase da captura, registrar o conjunto de ações tomadas pelo usuário e, durante a fase de repro-

dução, que as ações executadas anteriormente possam ser automaticamente repetidas, sem a intervenção do usuário. Ou seja, uma ferramenta de *capture/playback* facilita a criação desses *scripts*, gravando de forma automática as ações que o testador realiza por meio da interface. Esses *scripts* podem ser executados repetidamente e auxiliarem nos testes de regressão.

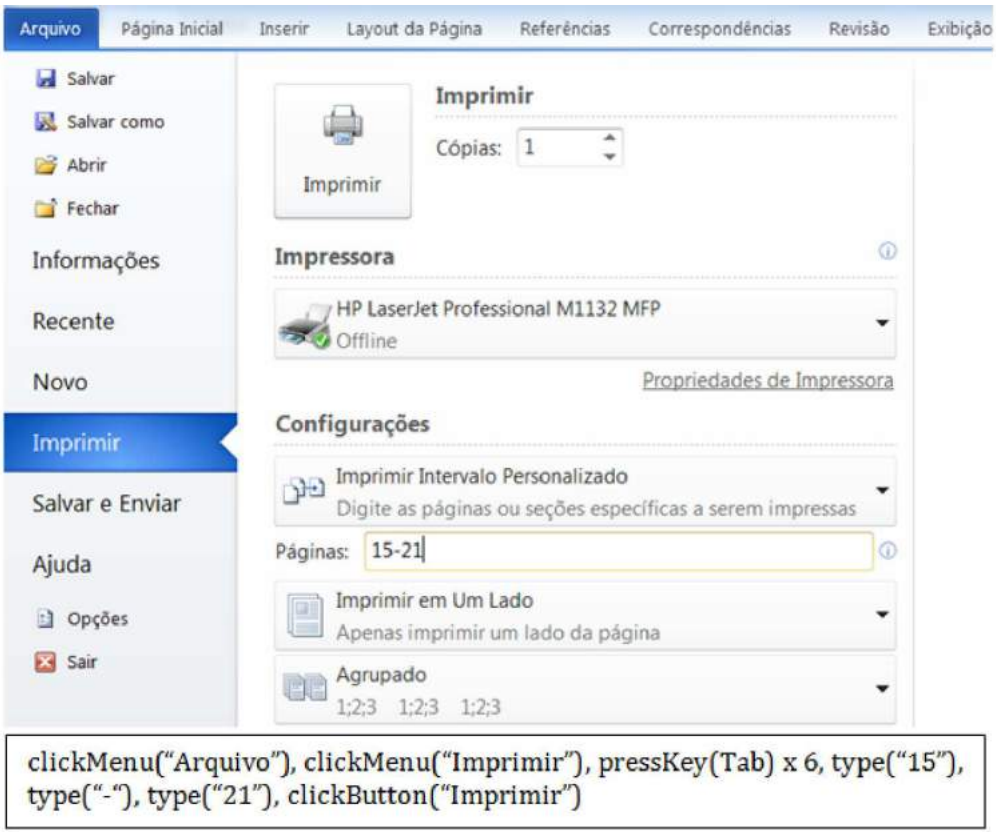


Figura 1.1: *Exemplo de Sequência de Entrada – Microsoft Word.*

Devido ao fato de que a interface sofre várias alterações ao longo do processo de desenvolvimento, muitos destes *scripts* serão invalidados, uma vez que estão diretamente relacionados e dependentes do nome ou da posição dos *widgets*, que podem ser modificados ou até mesmo removidos. Isso significa que os *scripts* devem ser sempre atualizados, consequentemente, elevando o custo dessa atividade (MEMON, 2001). Diante dessas dificuldades, as técnicas para a geração automática de dados de teste são bastante desejáveis.

Em geral, para abstrair a interface do usuário e permitir a geração de dados de teste, constrói-se um modelo para representar o fluxo de interação que a UI possibilita, sendo esse modelo variável. Existem três tipos de modelos mais utilizados para o teste baseado em UI (MBGT – *Base Model Gui Testing*): o modelo baseado em estados, o modelo baseado em eventos e engenharia reversa dinâmica.

O modelo baseado no estado é o mais explorado no cenário acadêmico (AHO et al., 2015) tendo com ideia principal representar o comportamento de uma aplicação de um nível de UI como uma *Finite-State-Machine* (FSM).Existem aplicações em alguns contextos, por exemplo, a *GUI Driver* (AHO; MENZ; RATY, 2011) e *GuiTam* (MIAO; YANG, 2010a) para aplicações JAVA GUI e *AndroidRipper* (AMALFITANO et al., 2012a) para aplicativos *Android*.

Outro formato popular para extração de modelos GUI é o modelo baseado em eventos. Memon et al. (2013) publicaram extensivamente sua pesquisa sobre GUI *Ripping*, uma técnica para extrair dinamicamente modelos baseados em eventos de aplicações GUI para fins de automação de teste.

A mais recente abordagem de extração do modelo UI é baseada em engenharia reversa dinâmica, ou seja, é feita a execução do aplicativo e se observa o comportamento em tempo de execução da UI. O grande desafio é passar automaticamente pela UI fornecendo dados significativos para os campos de entrada requisitados, como usuário e senha válidos para uma tela de *login* sem instruções predefinidas do usuário (AHO; MENZ; RATY, 2011).

Uma forma de lidar com a tarefa de gerar automaticamente dados de teste é representar o cenário de teste como um problema de otimização perante os tipos de modelos citados. A ideia é definir um critério de qualidade ou função de *fitness* que busquem dados de teste que maximizem esta função. Uma vez que o espaço de busca de todos os dados de teste possíveis é grande e, muitas vezes, tem uma estrutura complexa, podem ser exploradas técnicas de meta-heurística. Algumas técnicas de meta-heurística também já foram aplicadas no contexto de teste GUI. Por exemplo, o trabalho de Huang, Cohen e Memon (2010) usou algoritmo genético para corrigir conjuntos de teste inválidos. O trabalho consiste em duas etapas: gerar e, posteriormente, reparar o conjunto de teste caso contenha sequências inviáveis, que são impossíveis de serem executadas.

Huang, Cohen e Memon (2010) utilizou um modelo aproximado da GUI chamado de EFG (*Event Flow Graph*). Um EFG é um grafo direcionado cujos nós são as ações que o usuário pode executar (cliques por exemplo, sobre os itens de menu) e a transição entre a ação x e y significa: y está disponível após a execução de x. A Figura 1.2 mostra um EFG para o menu principal de um aplicativo GUI qualquer. Os nós correspondem a cliques em opções do menu e atravessando as arestas desse gráfico pode-se gerar sequências de ações (operações). Por exemplo, ao clicar no menu “Ajuda”, um menu secundário aparece contendo a opção “Sobre”, que por sua vez pode ser clicado.

Na primeira etapa, a fim de gerar um conjunto de teste inicial, o trabalho tenta identificar sequências válidas dentro do EFG. Uma vez que o EFG é apenas uma aproximação da GUI, esse conjunto de testes iniciais contém sequências de entrada inviáveis. Por exemplo: Na Figura 1.2 pode-se gerar $s = (Editar; Colar)$. No entanto, como na mai-

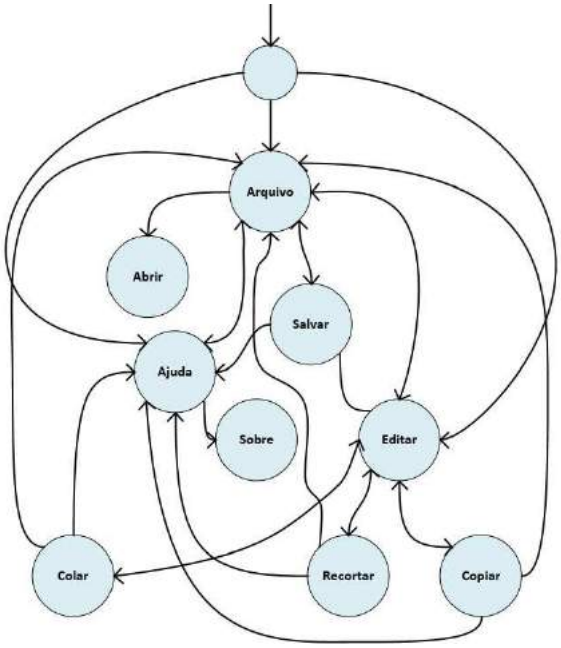


Figura 1.2: *Um Exemplo de EFG.*

oria das aplicações algumas entradas de menu são invisíveis, pelo menos até que uma operação de cópia ou recorte tenha ocorrido, a execução de s é impossível. Durante a segunda etapa são identificadas e descartadas as sequências inviáveis, sendo usado um algoritmo genético que utiliza o EFG para gerar novas sequências para compensar as que foram descartadas, sendo que as sequências inviáveis são penalizadas com um valor estático.

Além disso, as pesquisas ainda deixam alguns problemas em aberto (HUANG; COHEN; MEMON, 2010):

1. A enorme quantidade de sequências possíveis a partir de cada estado, ou seja, em cada estado existem muitas ações alternativas o que conduzem a um espaço de busca excepcionalmente grande. Além disso, é computacionalmente dispendiosos gerar e avaliar as sequências, visto que o software precisa ser iniciado e todas as ações da sequência precisam ser executadas. Isso requer algoritmos eficientes que explorem o espaço de busca de forma inteligente para encontrar as sequências ideais;
2. A falta de critérios de qualidade para caracterizar se uma determinada sequência tem qualidade;
3. As dificuldades das técnicas gerarem entradas para as aplicações que explorem cliques em botões e operações de arrastar e soltar componentes:
   (a) Mapear a UI para determinar os *widgets* visíveis e suas propriedades. Por exemplo, a posição dos componentes como botões e itens de menu;

(b) Derivar um conjunto de ações permitidas em cada fase de execução. Por exemplo, um botão visível não habilitado se seria clicável; e
(c) Executar, gravar e reproduzir essas informações mais tarde.

Os objetivos desse trabalho vão de encontro com alguns desses problemas e são apresentados na seção a seguir.

1.3 Objetivos

A visão desejada para o futuro perante a atividade de testes de aplicações UI seria que, se dada uma aplicação, fossem gerados automaticamente os dados de teste e os mesmos fossem executados e fosse produzida uma lista das possíveis falhas detectadas, sem intervenção humana. Porém, essa é uma tarefa ambiciosa. Este trabalho constitui um primeiro passo para contribuir na realização dessa tarefa, propondo um modelo de representação de WUI que traz benefícios em relação às representações hoje existentes na literatura. Também é proposto um algoritmo genérico para geração de dados de teste por meio da WUI de uma aplicação, contribuindo na automação do processo da atividade de teste.

Para alcançar este objetivo geral, alguns objetivos específicos foram determinados:

1. Identificar o estado da arte de testes UI;
2. Descrever os principais trabalhos encontrados na literatura, comparando e identificando lacunas e ferramentas automatizadas, dentro do contexto de teste UI.
3. Perante uma aplicação de software que use WUI, identificar e definir uma técnica para geração de dados de teste de forma automatizada;
4. Estudar e definir um modelo de representação da interface gráfica que possa ser gerado de forma automática ou semi-automática, e sirva como base para a geração de dados de teste, com ênfase em WUI.

1.4 Metodologia

Para atingir o objetivo geral e os objetivos específicos, apresentados na Seção 1.3, foi aplicada uma sequência de ações. Para o objetivo citado no Item 1, foi aplicado um mapeamento sistemático que é detalhado no Capítulo 4. Para o Item 2, foi feito um levantamento dos principais trabalhos e lacunas para teste UI e, perante eles, feito uma análise das principais ferramentas disponíveis na literatura. As descrições dessas lacunas e dessas ferramentas são apresentadas na Seção 4.4.

Para definição de uma técnica para geração de dados de teste de forma automatizada, Item 3, foi implementado um algoritmo genético, que é apresentado no Capítulo 5. O uso dessa meta-heurística justifica-se porque inicialmente, como objetivo principal desse trabalho, se pretendia trabalhar com a melhoria de algumas ferramentas existentes. Para isso, estudaram-se técnicas da área de pesquisa dentro da SBSE – *Search-based Software Engineering*, denominada SBST – *Search-Based Software Testing*, que foca à aplicação de técnicas de otimização matemática, na resolução de problemas no contexto da atividade de teste. Os estudos exploratórios, porém, demosntram que as ferramentas encontradas apresentaram muitas limitações, que são comentadas nas Seções 4.4.2 e 4.4.3. A principal delas é decorrente do modelo de abstração da UI utilizado. Frente a essa limitação, optou-se por concentrar esforços na proposição de um novo modelo para representar as formas de interação presentes, principalmente, no contexto de WUI. Assim sendo, para atingir o objetivo do Item 4, foi proposto um modelo de representação de WUI que é descrito no Capítulo 6.

1.5 Organização do Trabalho

Este capítulo apresenta o contexto no qual este trabalho se insere, as motivações para a sua realização, metodologia aplicada e seus objetivos. No Capítulo 2 são descritas as fases da atividade de teste, as principais técnicas e critérios de teste existentes e conceitos para geração automática de dados de teste. No Capítulo 3 são formalizados e exemplificados conceitos para aplicação do teste GUI e WUI. No Capítulo 4 são detalhadas as fases da aplicação do mapeamento sistemático e a descrição de alguns trabalhos e ferramentas, dentro do contexto dessa tese. No Capítulo 5 é descrita a representação do algoritmo genético proposto para geração automática de dados de teste WUI e um estudo exploratório para validação desse algoritmo. O modelo proposto para representação de WUI é formalizado no Capítulo 6. As conclusões, contribuições e trabalhos futuros são pontuados no Capítulo 7. Em seguida são relacionadas as bibliografias utilizadas para escrita dessa tese.

CAPÍTULO 2

Atividade de Teste de Software

2.1 Considerações Iniciais

Neste capítulo são apresentados alguns conceitos envolvendo a atividade de teste. Inicialmente, são feitas considerações a respeito da importância do teste durante o processo de desenvolvimento, em seguida são apresentadas as fases do teste e as principais técnicas e critérios que podem ser utilizadas em cada uma delas. Finalmente são apresentados conceitos para automatização dessa atividade e algumas meta-heurísticas utilizadas, na literatura, com esse objetivo.

2.2 Terminologias e Conceitos Básicos

Dada a grande importância do teste de software, esforços vêm sendo feitos para padronizar algumas terminologias comumente utilizadas. O padrão IEEE 24765-2010 (IEEE, 2010) define alguns termos e conceitos:

1. Dado de Teste (*Test Data*): é o dado enviado ao produto em teste. Corresponde ao dado de entrada fornecido à interface pública do produto em teste durante a execução do caso de teste;
2. Caso de Teste (*Test Case*): conjunto de dado de teste, condições de execução e resultados esperados e elaborados para atingir um objetivo específico de teste para verificar a conformidade com um determinado requisito;
3. Produto em Teste ou Sistema em Teste (*Product Under Test*, *System Under Test* ou SUT): sistema, subsistema ou componente em teste;
4. Conjunto de Teste (*Test Suite*): coleção de um ou mais casos de teste de um produto em teste;
5. Critério de Teste (*Test Criteria*): estabelece os requisitos que o conjunto de teste tem de atender para passar no teste. O critério de aceitação (*acceptance criterion*) de um usuário ou interessado (stakeholder) e as regras de decisão de que um sistema passou ou falhou nos testes (*pass/fail criterion*) são exemplos de critério de teste;

6. Engano (*Mistake*): ação humana que produz um resultado incorreto, como por exemplo, uma ação incorreta tomada pelo programador;
7. Defeito (*Fault*): passo, processo ou definição de dado incorreto, como por exemplo, uma instrução ou comando incorreto;
8. Erro (*Error*)[1]: diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro; e
9. Falha (*Failure*): produção de uma saída incorreta com relação à especificação. Um defeito pode levar a uma falha.

Para Maldonado et al. (2004), de uma forma geral, os defeitos são classificados em: **defeitos computacionais** – o defeito provoca uma computação incorreta, mas o caminho executado (sequências de comandos) é igual ao caminho esperado; e **defeitos de domínio** – o caminho efetivamente executado é diferente do caminho esperado, ou seja, um caminho errado é selecionado. Uma representação do relacionamento em defeito, erro e falha é mostrada na Figura 2.1.

Figura 2.1: *Defeitos X Erros X Falha (extraída de Delamaro, Maldonado e Jino (2007)).*

2.3 Etapas e Fases para Aplicação dos Testes

O processo de desenvolvimento de software envolve uma série de atividades nas quais, apesar das técnicas, critérios e ferramentas empregadas, defeitos no produto tendem a permanecer (MALDONADO et al., 2004). Com isso, a atividade de teste é de fundamental importância para a identificação e eliminação de defeitos remanescentes, representando a última revisão da especificação, projeto e codificação (PRESSMAN; MAXIM, 2014). Segundo Myers e Sandler (2004), a atividade de teste é o processo de

[1]Neste trabalho os termos "erro" e "defeito" serão utilizados como sinônimos para representar a causa de uma falha.

executar um programa com a intenção de encontrar uma falha; um bom caso de teste é aquele que tem alta probabilidade de revelar falhas e um teste bem sucedido é aquele que detecta a existência de um defeito ainda não descoberto.

A realização dos testes é composta das seguintes etapas: construção dos casos de teste, execução do programa P com esses casos de teste e análise do comportamento de P a fim de determinar se o mesmo está correto ou não. Tal procedimento se repete até que o testador tenha confiança de que P se comporta conforme o esperado com o mínimo de falhas possível ou até que uma falha seja descoberta (SOUZA, 1996).

Segundo Pressman e Maxim (2014), a atividade de teste deve ser conduzida em três fases: teste de unidade, teste de integração e teste de sistema. O teste de unidade concentra esforços na verificação da menor parte de projeto de software chamada de módulo ou unidade. O teste de integração é uma atividade sistemática para a construção da estrutura de programa, visando a descobrir erros associados às interfaces de comunicação entre as unidades. O objetivo é, a partir das unidades testadas individualmente, construir a estrutura de programa que foi determinada pelo projeto. O teste de sistema, realizado após a integração do mesmo, visa a identificar erros de funções e características de desempenho que não estejam de acordo com a especificação.

Um ponto crucial na atividade de teste é o projeto e a avaliação da qualidade de um conjunto de teste utilizado para testar um determinado programa P. Um caso de teste consiste de um par ordenado $(d, S(d))$, no qual d é um elemento de um dado domínio D $(d \in D)$ e $S(d)$ é a saída esperada para uma dada função, em relação à especificação, quando d é utilizado como entrada. Uma verificação completa de P poderia ser obtida testando P com um conjunto de casos de teste T que inclui todos os elementos do domínio. Entretanto, como geralmente o conjunto de elementos do domínio é infinito ou muito grande, torna-se praticamente impossível testar todos os elementos do domínio e, dessa forma, deve ser escolhido um subconjunto para ser utilizado para os testes. Para Myers e Sandler (2004), essa escolha é um dos pontos críticos da atividade de teste, pois o programa, para ser demonstrado correto, deveria ser exercitado com todos os valores possíveis do domínio de entrada. Porém, sabe-se que o teste exaustivo é impraticável devido a restrições de tempo e custo para realizá-lo, sendo necessário determinar um conjunto de casos de teste que seja eficaz em encontrar erros e cuja cardinalidade seja reduzida. Além disso, a atividade de teste é permeada por uma série de limitações (RAPPS; WEYUKER, 1985; HOWDEN, 1987b). Em geral, os seguintes problemas são indecidíveis: dado dois programas, se eles são equivalentes; dados duas sequências de comandos (caminhos) de um programa, ou de programas diferentes, se eles computam a mesma função; e dado um caminho, se ele e executável ou não, ou seja, se existe um conjunto de dados de entrada que leva à execução desse caminho. Outra limitação fundamental é a correção coincidente, ou seja, o programa pode apresentar,

coincidentemente, um resultado correto para um item particular de um dado $d \in D$, ou seja, um particular item de dado ser executado, satisfazer a um requisito de teste e não revelar a presença do erro. Desta forma, para selecionar e avaliar conjuntos de teste, é fundamental a utilização de critérios de teste que também auxiliam o testador a decidir quando parar os testes.

2.4 Técnicas e Critérios de Teste

Na tentativa de reduzir os custos associados ao teste, é fundamental a aplicação de técnicas que indiquem como testar o software e de critérios que respondam quando parar os testes, de forma que essa atividade possa ser conduzida de modo planejado e sistemático (DEMILLO, 1980). Já os critérios de teste são elaborados com o objetivo de fornecer uma maneira sistemática e rigorosa para selecionar um subconjunto do domínio de entrada e ainda assim ser eficaz para revelar a presença de defeitos, respeitando as restrições de tempo e custo associados a um projeto de software. Esses critérios são classificados em três técnicas de testes: Técnica Funcional, Técnica Estrutural e a Técnica Baseada em Defeitos. A diferença entre as três técnicas é a origem da informação usada para avaliar ou para construir conjuntos de teste (DELAMARO; MALDONADO; JINO, 2007).

Segundo Pressman e Maxim (2014), nenhuma das técnicas de teste é completa, no sentido que nenhuma delas é, em geral, suficiente para garantir a qualidade da atividade de teste. Essas diferentes técnicas são complementares e devem ser aplicadas em conjunto para assegurar um teste de boa qualidade.

Apesar das limitações da atividade de teste, sua aplicação de maneira sistematizada e bem planejada pode garantir ao software algumas características mínimas que são importantes no estabelecimento da qualidade do produto e relevantes também para o seu processo de evolução. Além disso, é importante lembrar que o teste é, em geral, apenas uma atividade de validação e que sua utilização isolada não é suficiente para alcançar um produto de boa qualidade. Outras técnicas, tais como inspeção, "*walkthrough*" e verificação formal, devem ser utilizadas em conjunto com a atividade de teste (PRESSMAN; MAXIM, 2014).

2.4.1 Teste Funcional

Segundo Delamaro, Maldonado e Jino (2007), o teste funcional é uma técnica utilizada para se projetarem casos de teste, na qual o programa ou sistema é considerado uma caixa preta e, para testá-lo, são fornecidas entradas e avaliadas as saídas geradas para verificar se estão em conformidade com os objetivos especificados. Os detalhes de

implementação quando se usa essa técnica não são considerados e o software é avaliado segundo o ponto de vista do usuário. Para isto, Coward (1988) distingue no teste funcional dois passos principais: identificar as funções esperadas do software e criar casos de teste que chequem a realização dessas funções. Com isso, é de suma importância a qualidade das especificações de software para este tipo de teste, visto que funções em questão são identificadas a partir delas (DEMILLO, 1987). Para Pressman e Maxim (2014), o teste de caixa preta é uma abordagem que procura revelar falhas das seguintes categorias: funções incorretas ou ausentes; interface; falhas nas estruturas de dados ou no acesso a banco de dados externos; desempenho; e inicialização e término.

Uma grande vantagem da técnica funcional e de seus critérios, viabilizando a sua larga utilização na validação de produtos de software, é que necessitam apenas da especificação do produto para derivar os requisitos de teste. Desse modo, é possível aplicá-los a qualquer programa, seja ele procedimental, orientado a objetos ou a componentes de software e em qualquer fase de teste, uma vez que o código-fonte não é usado como fonte de requisitos de teste. Além disso, seus critérios são aplicados da mesma forma, independentemente da fase de teste e, por esse motivo, consiste no principal tipo de teste realizado na fase do teste de sistema. Por outro lado, de acordo com Roper (1994), como os critérios funcionais se baseiam apenas na especificação, eles não podem assegurar que partes críticas e essenciais do código foram executadas durante os testes. Dentre os diversos critérios da técnica funcional existentes destacam-se: Particionamento em Classes de Equivalência, Análise do Valor Limite (BVA – *Boundary Value Analysis*) e Grafo de Causa-Efeito.

O critério Particionamento em Classes de Equivalência tem como objetivo apoiar a determinação de um subconjunto do domínio de entrada para ser utilizado nos testes, o particionamento em classes de equivalência sugere particionar o domínio de entrada de um programa em um número finito de classes de equivalência de tal forma que se possa assumir que o teste de um valor representativo de cada classe é equivalente ao teste de qualquer outro valor da classe. Isto é, assume-se que, se um caso de teste pertencente a uma classe de equivalência detectar um erro, todos os outros casos de teste na mesma classe de equivalência devem encontrar o mesmo erro. Inversamente, se um caso de teste não detectar um erro, espera-se que nenhum outro caso de teste na mesma classe de equivalência também o detecte. Além disso, alguns autores consideram não apenas do domínio de entrada, mas também a partição do domínio de saída, procurando identificar nestas possíveis alternativas novas classes de equivalência no domínio de entrada (ROPER, 1994; COPELAND, 2003). Segundo Myers e Sandler (2004), o projeto de casos de teste com a utilização desse critério é conduzido em dois passos:

1. Identificação das classes de equivalência: a partir da especificação do programa ou sistema, as classes de equivalência são identificadas tomando-se cada condição

de entrada e particionando-as em classes de equivalência válidas e inválidas. Caso seja observado que as classes de equivalência se sobrepõem ou que os elementos de uma mesma classe não devem se comportar da mesma maneira, elas devem ser reduzidas a fim de separá-las e torná-las disjuntas (DELAMARO; MALDONADO; JINO, 2007); e

2. Identificação dos casos de teste: uma vez identificadas as classes de equivalência, devem-se determinar os casos de teste, escolhendo-se um elemento de cada classe, de forma que cada novo caso de teste cubra o maior número de classes válidas possível. Para as classes inválidas devem ser gerados casos de teste exclusivos, uma vez que um elemento de uma classe inválida pode mascarar a validação do elemento de outra classe inválida (COPELAND, 2003).

De acordo com Delamaro, Maldonado e Jino (2007), a força deste critério está na redução que ele possibilita no tamanho do domínio de entrada e na criação de dados de teste baseados unicamente na especificação, sendo adequado para aplicações em que as variáveis de entrada podem ser identificadas com facilidade e assumem valores específicos. No entanto, Delamaro, Maldonado e Jino (2007) ressaltam que o critério não é tão facilmente aplicável quando o domínio de entrada é simples e seu processamento é complexo pois, embora a especificação possa sugerir que um grupo de dados seja processado de forma idêntica, na prática isso pode não acontecer. Além disso, a técnica não fornece diretrizes para a determinação dos dados de teste e para encontrar combinações entre eles que permitam cobrir as classes de equivalência de maneira mais eficiente (ROPER, 1994).

Com o critério Análise do Valor Limite a experiência mostra que casos de teste que exploram condições limites têm uma maior probabilidade de encontrar erros. Ou seja, os valores que estão exatamente sobre ou imediatamente acima ou abaixo dos limites das classes de equivalência têm maior probabilidade de revelar defeitos (MYERS; SANDLER, 2004). Sendo assim, a análise do valor limite é uma critério de projeto de casos de teste que completa o particionamento em classes de equivalência (PRESSMAN; MAXIM, 2014) pois, ao invés de os dados de teste serem escolhidos aleatoriamente em uma classe de equivalência, eles devem ser selecionados para que os limites de cada classe de equivalência sejam explorados. Ainda segundo Myers e Sandler (2004), além da escolha seletiva dos dados de teste, o outro ponto que distingue esse critério do critério de particionamento em classes de equivalência é a observação do domínio de saída. De acordo com Pressman e Maxim (2014), as diretrizes para a aplicação desse critério são semelhantes em muitos aspectos às fornecidas para o particionamento em classes de equivalência:

1. Se uma condição de entrada especifica um intervalo limitado pelos valores *a* e *b*, casos de teste devem ser projetados com os valores *a* e *b*, e imediatamente acima e imediatamente abaixo de *a* e *b*;
2. Se uma condição de entrada especifica vários valores, casos de teste devem ser desenvolvidos para exercitar os números mínimo e máximo. Valores imediatamente acima e imediatamente abaixo do mínimo e do máximo também são testados;
3. Aplique as diretrizes 1 e 2 às condições de saída; e
4. Se as estruturas de dados internas do programa têm limites prescritos, certifique-se de projetar um caso de teste para exercitar a estrutura de dados no seu limite.

Este critério, como dito anteriormente, é muito similar ao critério de particionamento em classes de equivalência com relação a vantagens e desvantagens de uso. No entanto, segundo Myers e Sandler (2004), se aplicado corretamente, é um dos critérios mais úteis para o projeto de casos de teste.

Uma das limitações dos critérios da técnica funcional apresentada até o momento é que eles não exploram combinações dos dados de entrada, pois o teste de combinações de entrada não é uma tarefa simples, já que o número de combinações geralmente é muito grande. Para minimizar esta dificuldade, surgiu o critério chamado Grafo de Causa-Efeito. Ele define uma maneira sistemática de seleção de um conjunto de casos de teste que explora ambiguidades e incompletude nas especificações. Como forma de derivar os casos de teste, este critério utiliza um grafo que é uma linguagem formal na qual a especificação é traduzida (MYERS; SANDLER, 2004). O processo de aplicação deste critério pode ser resumido nos seguintes passos:

1. Dividir a especificação do software em partes, pois a construção do grafo para grandes especificação torna-se bastante complexa;
2. Identificar as causas e efeitos na especificação. As causas correspondem às entradas, estímulos, ou qualquer evento que provoque uma resposta do produto em teste e os efeitos correspondem às saídas, mudanças de estado do sistema ou qualquer resposta observável. Uma vez identificados, a cada um deve ser atribuído um único número;
3. Analisar a semântica da especificação e transformar em um grafo booleano – o grafo causa-efeito – que liga as causas e os efeitos;
4. Adicionar anotações no grafo, que descrevem combinações das causas e efeito, impossíveis por causa de restrições semânticas ou do ambiente;
5. Converter o grafo em uma tabela de decisão, na qual cada coluna representa um caso de teste; e
6. Converter as colunas da tabela de decisão em casos de teste.

A utilização deste critério se torna complexa quando é necessário construir o grafo booleano para um número elevado de causas e efeitos.

Em geral, o teste funcional é uma técnica de validação de programas na qual os casos de teste são gerados a partir da especificação dos requisitos, tornando-se uma técnica sujeita às inconsistências que podem ocorrer na especificação (DEMILLO, 1987). Outro problema encontrado com a utilização dessa técnica é a dificuldade de quantificar a atividade de teste, visto que não se pode garantir que partes essenciais ou críticas do programa sejam executadas.

2.4.2 Teste Estrutural

Essa técnica de teste, também conhecida como teste de caixa branca, estabelece requisitos de teste com base em uma dada implementação, requerendo a execução de partes ou de componentes elementares do programa (MYERS; SANDLER, 2004; PRESSMAN; MAXIM, 2014). Para isso, baseia-se no conhecimento da estrutura interna do programa, e os aspectos de implementação são fundamentais para a geração/seleção dos casos de teste associados.

Em geral, a maioria dos critérios da técnica estrutural utiliza uma representação de programa conhecida como "Grafo de Fluxo de Controle" (GFC) ou "Grafo de Programa", que mostra o fluxo lógico do programa.

Um GFC G, onde $G = (N, E, s)$, é um grafo dirigido que consiste de um conjunto N de nós, um conjunto E de arestas dirigidas e um nó de entrada s. Os nós representam comandos ou uma coleção de comandos sequenciais (blocos de comandos), e os arcos ou arestas representam o fluxo de controle. O grafo de fluxo de controle possui um nó de entrada e um ou mais nós de saída nos quais a computação começa e termina, respectivamente. O nó de entrada não possui nenhuma aresta de entrada, ou seja, não possui antecessor. Por outro lado, os nós de saída não possuem arestas de saída, ou seja, não possuem sucessores (ZHU; HALL; MAY, 1997).

Um bloco de comandos (ou bloco de instruções) consiste em um conjunto de comandos de uma determinada unidade, de modo que, quando o primeiro comando do bloco é executado, os outros comandos do mesmo bloco também são executados sequencialmente de acordo com a ordem estabelecida. Assim todos os comandos de um bloco têm um único predecessor e um único sucessor, com exceção do primeiro bloco que não tem um predecessor e do último que não tem um sucessor. Além disso, o primeiro comando de um bloco é o único comando que pode ser executado depois da execução do último comando do bloco anterior. Cada bloco corresponde a um nó e a transferência de controle de um bloco para outro é representada por arestas dirigidas entre os nós (RAPPS; WEYUKER, 1985; ZHU; HALL; MAY, 1997).

Um caminho é uma sequência finita de nós $n_1, n_2, ..., n_k$, $k \geqslant 2$, tal que existe uma aresta de n_i para n_{i+1} para $i = 1, 2, ..., k-1$. Um caminho completo é um caminho no qual o primeiro nó é o nó de entrada e o último nó é um nó de saída do grafo G (RAPPS; WEYUKER, 1985). Um caminho é um caminho simples se todos os nós que compõem esse caminho, exceto possivelmente o primeiro e o último, são distintos. Um caminho independente é qualquer caminho ao longo do programa que introduz pelo menos um novo nó ou uma nova aresta. Um caminho livre-de-laço é um caminho em que todos os nós são distintos, ou seja, nenhum nó aparece mais que uma vez. Um caminho é um caminho livre-de-iteração-de-laço se ele não contém o mesmo nó mais que duas vezes (LINNENKUGEL; MüLLERBURG, 1990). No teste estrutural existem também os caminhos não executáveis. Um caminho não executável é um caminho do grafo impossível de ser coberto para qualquer elemento do domínio de entrada. Isso acontece quando as condições lógicas que deveriam ser satisfeitas para que a sequência de nós do caminho fosse executada são contraditórias (HOWDEN, 1987a).

Desta forma, a partir do grafo de fluxo de controle, os caminhos lógicos do software são testados, fornecendo-se casos de teste que põem à prova tanto conjuntos específicos de condições e/ou laços bem como pares de definições e usos de variáveis. Os critérios pertencentes à técnica estrutural são classificados com base na complexidade, no fluxo de controle e no fluxo de dados (WEYUKER, 1984; MALDONADO, 1991; PRESSMAN; MAXIM, 2014). A técnica estrutural apresenta uma série de limitações e desvantagens decorrentes das limitações inerentes à atividade de teste que podem introduzir sérios problemas na automatização do processo de validação de software (MALDONADO, 1991). Independentemente dessas desvantagens, essa técnica é vista como complementar às demais técnicas de teste existentes, uma vez que cobre classes distintas de defeitos (MYERS; SANDLER, 2004; PRESSMAN; MAXIM, 2014). Além disso, as informações obtidas pela aplicação de critérios estruturais são consideradas relevantes para as atividades de manutenção, depuração e avaliação da confiabilidade de software (CHAIM, 2001; DELAMARO; MALDONADO; JINO, 2007; SOUZA, 2012). Como exemplos de critérios de teste estrutural, têm-se os baseados (PRESSMAN; MAXIM, 2014): na Complexidade, no Fluxo de Controle e no Fluxo de Dados.

O critérios baseados na Complexidade utilizam informações sobre a complexidade do programa para determinar os requisitos de teste. Um critério bastante conhecido desta classe é o critério de McCabe (1976), que utiliza a complexidade ciclomática para derivar os requisitos de teste. Essencialmente, esse critério requer que seja executado um conjunto de caminhos linearmente independentes do grafo de programa (PRESSMAN; MAXIM, 2014).

A complexidade ciclomática é uma métrica de software que fornece uma medida quantitativa da complexidade lógica de um programa. Quando usada no contexto do

método de teste de caminho básico, o valor calculado para a complexidade ciclomática define o número de caminhos independentes no conjunto-base de um programa e fornece um limite superior para a quantidade de testes que deve ser conduzida para garantir que todos os comandos sejam executados pelo menos uma vez (PRESSMAN; MAXIM, 2014). Ela tem fundamentação na teoria dos grafos e é calculada por uma das seguintes formas possíveis:

1. O número de regiões em um GFC. Uma região pode ser informalmente descrita como uma área incluída no plano do grafo. O número de regiões é computado contando-se todas as áreas delimitadas e a área não delimitada fora do grafo; ou
2. $V(G) = E - N + 2$, tal que E é o número de arestas e N é o número de nós do GFC G; ou
3. $V(G) = PN + 1$, tal que PN é o número de nós predicativos contido no GFC G.

Com isso, a complexidade ciclomática é uma métrica útil para previsão dos módulos que provavelmente sejam propensos a erro. Ela pode ser usada tanto para o planejamento de teste quanto para o projeto de casos de teste.

Já os critérios baseados em Fluxo de Controle utilizam apenas características de controle de execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias (MYERS; SANDLER, 2004; PRESSMAN; MAXIM, 2014). Os mais conhecidos são os critérios:

1. Todos-Nós – exige que a execução do programa passe, ao menos uma vez, por cada vértice do GFC, ou seja, que cada comando do programa seja executado pelo menos uma vez;
2. Todos-Arcos – requer que cada arco do grafo, isto é, cada desvio de fluxo de controle do programa seja exercitada pelo menos uma vez; e
3. Todos-Caminhos – requer que todos os possíveis caminhos do programa sejam exercitados o que, em geral, é impraticável.

Para Delamaro, Maldonado e Jino (2007) a cobertura do critério Todos-Nós é o mínimo esperado de uma boa atividade de teste, pois, do contrário, o programa testado é entregue sem a certeza de que todos os comandos presentes foram executados ao menos uma vez. Além disso, outro ponto a se destacar é que, apesar de desejável, a cobertura do critério Todos-Caminhos de um programa é, na maioria dos casos, uma tarefa impraticável por causa da quantidade de requisitos de teste gerados. Esse problema foi uma das motivações para o surgimento dos critérios baseados em Fluxo de Dados.

Os critérios baseados em Fluxo de Dados utilizam informações do fluxo de dados do programa para derivar os requisitos de teste. Esses critérios selecionam caminhos de teste de um programa de acordo com as localizações das definições (pontos em que as

variáveis recebem um valor) e usos (pontos em que os valores das variáveis são utilizados) de variáveis no programa.

Para que fosse possível derivar os requisitos de teste exigidos por tais critérios, seria necessário estender o GFC para armazenar informações a respeito do fluxo de dados do programa. Essa extensão do GFC, proposta por Rapps e Weyuker (1982), é chamada de Grafo Def-Uso (*Def-Use Graph*) e permite que sejam exploradas as interações que envolvem definições de variáveis e os subsequentes usos dessas variáveis. A definição (def) de uma variável ocorre toda vez que um valor é atribuído a ela. O uso de uma variável, por sua vez, pode ser de dois tipos: quando a variável é usada em uma computação, diz-se que seu uso é computacional (c-uso) e quando a variável é usada em uma condição, diz-se que seu uso é predicativo (p-uso). Outro conceito importante é o par def-uso, que se refere a um par de definição e subsequente c-uso ou p-uso de uma variável. Um caminho livre de definição com relação a uma variável x do nó i ao nó j é um caminho $(i, n_1, ..., n_m, j)$ para $m \geqslant 0$, no qual não há definições de x nos nós $n_1, ..., n_m$.

Para que as informações de definição e uso das variáveis sejam adicionadas ao grafo Def-Uso, cada nó i do grafo é associado aos conjuntos *c-uso* e def, e cada aresta (i, j) ao conjunto *p-uso*. $def(i)$ é um conjunto de variáveis definidas no nó i; $c\text{-}uso(i)$ é um conjunto de variáveis para as quais existem c-usos em i e $p\text{-}uso(i, j)$ é um conjunto de variáveis para as quais existem p-uso na aresta (i, j). Definem-se ainda outros conjuntos necessários para a construção do critério def-uso (RAPPS; WEYUKER, 1982). Considere um nó e uma variável x tal que $x \in def(i)$. Assim, Rapps e Weyuker (1982) definem:

1. $dcu(x;i)$ é o conjunto de todos os nós j tais que $x \in c\text{-}uso(j)$ e para os quais existe um caminho livre de definição com relação a x de i a j; e
2. $dpu(x;i)$ é o conjunto de arestas (j,k) tais que $x \in p\text{-}uso(j,k)$ e para as quais existe um caminho livre de definição com relação a x de i a (j,k).

Seja P um conjunto de caminhos completos para um grafo Def-Uso de um programa. Diz-se que um nó i está incluído em P se P contém um caminho $(n_1, ..., n_m)$ tal que $i = n_j$ para algum j, $1 \geqslant j \geqslant m$. Similarmente, uma aresta (i_1, i_2) está incluída em P se P contém um caminho $(n_1, ..., n_m)$ tal que $i_1 = n_j$ e $i_2 = n_{j+1}$ para algum j, $1 \geqslant j \geqslant m-1$. Um caminho $(i_1, ..., i_k)$ está incluído em P se P contém um caminho $(n_1, ..., n_m)$ tal que $i_1 = n_j, i_2 = n_{j+1}, ..., i_k = n_{j+k-1}$ para algum j, $1 \geqslant j \geqslant m-k+1$ (RAPPS; WEYUKER, 1982).

Baseando-se no trabalho de Rapps e Weyuker (1982), considere G um grafo Def-Uso e P um conjunto de caminhos completos de G. Dentre os critérios baseados em fluxo de dados, são definidos os seguintes:

1. Todas-Definições: P satisfaz o critério Todas-Definições se para cada nó i do grafo Def-Uso e para cada $x \in def(i)$, P inclui um caminho livre de definição com relação a x de i a algum elemento de $dcu(x,i)$ ou $dpu(x,i)$;
2. Todos-C-Usos: P satisfaz o critério Todos-C-Usos se para cada nó i do grafo Def-Uso e para cada $x \in def(i)$, P inclui um caminho livre de definição com relação a x de i a algum elemento de $dcu(x,i)$;
3. Todos-P-Usos: P satisfaz o critério Todos-C-Usos se para cada nó i do grafo Def-Uso e para cada $x \in def(i)$, P inclui um caminho livre de definição com relação a x de i a algum elemento de $dpu(x,i)$;
4. Todos-Usos: P satisfaz o critério Todos-Usos se para cada nó i do grafo Def-Uso e para cada $x \in def(i)$, P inclui um caminho livre de definição com relação a x de i a cada elemento de $dcu(x,i)$ e a cada elemento de $dpu(x,i)$; e
5. Todos-Caminhos-DU: P satisfaz o critério Todos-Caminhos-DU se para cada nó i do grafo Def-Uso e para cada $x \in def(i)$, P inclui cada caminho livre-de-laço e livre de definição com relação a x de i a cada elemento de $dpu(x,i)$ e a cada elemento de $dcu(x,i)$.

2.4.3 Teste Baseado em Defeitos

A técnica de Teste Baseado em Defeitos utiliza informações sobre os defeitos mais frequentes cometidos no processo de desenvolvimento de software e sobre os tipos específicos de defeitos que se desejam localizar (DEMILLO, 1987). A ênfase da técnica está nos defeitos que o programador ou projetista pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar a sua ocorrência. Semeadura de Erros (*Error Seeding*) e critérios baseados em Mutação são critérios típicos que se concentram em defeitos.

No critério Semeadura de Erros uma quantidade conhecida de defeitos é semeada artificialmente no programa. Após o teste, do número total de defeitos encontrados verificam-se quais são naturais e quais são artificiais. Usando estimativas de probabilidade, o número de defeitos reais remanescentes no programa pode ser calculado (BUDD, 1981). Segundo Budd (1981), dentre os problemas para a aplicação deste critério estão:

1. Os defeitos artificiais podem interagir com os naturais fazendo com que os defeitos naturais sejam "mascarados" pelos defeitos semeados;
2. Para obter-se um resultado estatístico não questionável, é necessário o uso de programas capazes de conter 10.000 defeitos ou mais; e
3. É preciso assumir que os defeitos estejam distribuídos pelo programa de maneira uniforme, o que, em geral, não é verdade. É comum programas reais apresentarem

longos trechos com códigos simples e poucos defeitos, e pequenos trechos com alta complexidade e alta concentração de defeitos.

Para os critérios de teste baseados de Mutação, utiliza-se um conjunto de programas ligeiramente modificados (mutantes) obtidos a partir do programa em teste. O objetivo é encontrar um conjunto de teste capaz de revelar as diferenças existentes entre o programa original e seus mutantes (DEMILLO, 1987).

Primeiramente o testador deve fornecer um programa P a ser testado e um conjunto de casos de teste T cuja adequação deseja-se avaliar. O programa é executado com T e, se apresentar resultados incorretos, então uma falha foi revelada e o teste termina. Caso contrário, o programa ainda pode conter defeitos que o conjunto T não conseguiu revelar. O programa P sofre então pequenas alterações, dando origem aos programas $P_1, P_2 ... P_n$, que são mutantes de P, diferindo deste apenas pela ocorrência de defeitos simples, ou seja, cada mutante contém apenas uma mutação. A seguir, os mutantes são executados com o mesmo conjunto de teste T. Os mutantes que apresentam resultados diversos de P, para algum caso de teste, é dito "morto". Os demais são considerados vivos e entre estes existem os mutantes equivalentes, para os quais não existe um valor do domínio de entrada que diferencie seu resultado de P.

O Teste de Mutação, aplicado em nível de unidade, é denominado Análise de Mutante e se mostra um critério efetivo para o teste da estrutura interna da unidade, mas não necessariamente para exercitar as interações entre unidades em um programa integrado (DELAMARO; MALDONADO; MATHUR, 2001). Com isso, surgiu a Mutação de Interface que insere perturbações nas conexões entre duas unidades, de modo que, para obter um conjunto de teste adequado, o testador deve criar casos de teste que exercitem essas conexões. Utilizando o mesmo raciocínio aplicado à Análise de Mutantes, casos de teste capazes de distinguir mutantes de interface também devem ser capazes de revelar grande parte dos defeitos de integração (DELAMARO; MALDONADO; MATHUR, 2001).

Considerando o critério Análise de Mutantes, dois outros critérios que podem ser derivados são a Mutação Fraca (*Weak Mutation*) (HOWDEN, 1982) e a Mutação Firme (*Firm Mutation*) (WOODWARD; HALEWOOD, 1988). A ideia básica dos critérios Mutação Fraca, Mutação Firme e Mutação Forte (como usa Howden (1982) para se referir à Análise de Mutantes) é a mesma, ou seja, uma pequena mudança no programa é realizada e os resultados da versão original são comparados com os da versão modificada. A diferença está no momento da criação do mutante e no momento em que se comparam os resultados obtidos para decidir se o mutante "morre" ou continua "vivo". Na Mutação Fraca, cria-se o mutante imediatamente antes da execução de um comando e os resultados são comparados logo após o término da execução do comando. Na Mutação Forte, geram-se os mutantes antes do início da execução do programa e comparam-se os resultados

após o término de sua execução. Já a Mutação Firme, definida como um critério entre a Mutação Fraca e a Mutação Forte, realiza alterações no programa e introduz pontos nos quais os estados do programa original e do programa mutante são comparados antes do final da execução (WOODWARD; HALEWOOD, 1988). A grande desvantagem desses critérios fica por conta do grande número de mutantes gerados, mesmo para aplicações simples.

Para ajudar com os problemas apresentados pela aplicação das técnicas e critérios de teste, a seguir são descritas formas para geração automática dos dados de teste, procurando, assim, diminuir o custo de aplicação.

2.5 Geração Automática de Dados de Teste

A escolha criteriosa dos casos de teste deve ser realizada a fim de se identificar aqueles com alta probabilidade de encontrar possíveis defeitos existentes dentro do prazo estabelecido para a entrega dos resultados em um projeto, por exemplo. Sendo assim, independentemente do critério de teste empregado, a geração de casos de teste corresponde a uma importante atividade dentro da fase de teste. Por sua vez, a geração de casos de teste compreende os esforços de se estabelecer os dados de entrada, que serão usados para execução do programa, e as respectivas saídas esperadas para cada entrada.

Para Roper (1994), teste é apenas amostragem, ou seja, como, em geral, o teste exaustivo não é viável, o testador seleciona uma amostra das possíveis entradas do produto a ser testado (dados de teste), identifica a saída esperada para essas entradas, criando os casos de teste, e o programa é executado com esses casos de teste para verificar se as saídas produzidas estão de acordo com as saídas esperadas. Esse é o ponto principal da atividade de teste e, em geral, é desempenhado pelo testador, devido a sua complexidade e indecidibilidade.

Em geral, a geração de dados de teste (GDT) é um problema indecidível, devido à complexidade, o tamanho de programas e o tamanho do domínio de entrada. Porém, algumas técnicas propostas investigam a utilização de meta-heurísticas para a resolução deste problema. Para isso, é preciso transformar o critério de teste desejado em uma função objetivo, que corresponde a uma representação matemática responsável por comparar e contrastar as soluções retornadas pela busca, a fim de alcançar uma meta global de busca. Desta forma, os tipos de funções objetivo que podem ser geradas são dependentes do critério escolhido, ou seja, transforma-se em função o objetivo da meta que dado critério estabelece.

Essas técnicas de busca meta-heurística, também chamadas na literatura de algoritmos de busca e otimização, são técnicas extraídas da área de Inteligência Artificial (IA) utilizadas em problemas de otimização. A resolução do problema é realizada automati-

camente por meio da utilização de métodos inteligentes, em que a obtenção de conhecimento ocorre durante a sua execução ou previamente, permitindo a adequação dos dados até que seja atingindo o grau de qualidade desejado. Algumas dessas meta-heurísticas são sucintamente descritas na Seção 2.5.2

Para garantir um sistema livre de defeitos, todo software deveria ser executado com todas as entradas possíveis de forma exaustiva, de modo que todas as possibilidades de execução do sistema fossem exploradas, antes de sua liberação, como representado pela Figura 2.2. Porém, como já citado anteriormente, o teste exaustivo é impraticável.

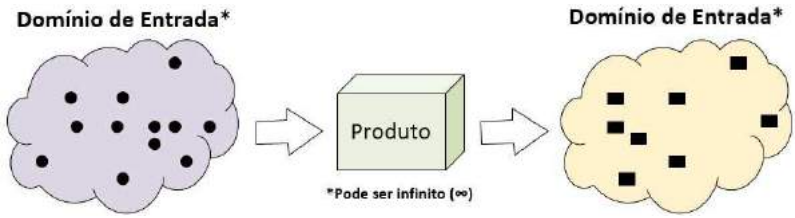


Figura 2.2: *Domínios de Entrada e Saída de Dado (adaptado de Machado, Vincenzi e Maldonado (2010)).*

O exemplo a seguir é uma adaptado do trabalho "*On the reliability of mechanisms*" descritos por Dijkstra (1970), no qual se demonstra a impossibilidade do teste exaustivo para a maioria dos programas e se define o famoso corolário: "o teste de programa somente pode ser utilizado para detectar a presença de defeitos, mas nunca para detectar a sua ausência".

Considere um simples método em JAVA que recebe de parâmetro dois argumentos de um tipo primitivo `double`, cada um com uma representação de 64 bits. Esse método tem claramente um domínio de entrada finito com 2^{128} elementos ($2^{64} * 2^{64}$) considerando todas as combinações possíveis. Supondo que esse método seja executado em uma máquina capaz de realizar 2^{40} instruções por segundo, que é compatível com a velocidade dos processadores atuais, o teste exaustivo desse método levaria ($\frac{2^{128}}{2^{40}} = 2^{88} \approx 10^{26}$) segundos para ser completado. Como um ano tem aproximadamente 10^7 segundos, a execução dos casos de teste terminariam em torno de $\frac{10^{26}}{10^7} = 10^{19}$ anos, claramente um prazo inviável de ser cumprido.

A partir do exemplo acima, pode-se observar que a grande dificuldade do teste consiste em identificar quais os melhores elementos a serem selecionados para executar o programa em teste de modo a detectar a maior quantidade de defeitos no menor tempo e no menor custo, ou seja, como criar os melhores dados de teste.

Para auxiliar a sistematizar a atividade de teste e orientar o testador se um software foi suficientemente testado, é que foram criados os critérios de teste; alguns citados na Seção 2.4. O principal objetivo desses critérios é subdividir o domínio de entrada em subdomínios, não necessariamente disjuntos, e exigir do testador que pontos

de cada subdomínio sejam selecionados (dados de teste). Mas quais pontos devem ser escolhidos? Em princípio, deveriam ser escolhidos aqueles com maior probabilidade de detectar a presença de defeitos no programa sendo testado. As estratégias empregadas para a geração de dados de teste são de fundamental importância, pois delas depende a eficácia dos dados de teste gerados.

A completa automatização para a geração automática de dados de teste é prejudicada devido a problemas, tais como: caminho ausente, caminhos não executáveis e equivalência de programas. Entretanto, mesmo na presença dessas limitações, várias pesquisas são realizadas nessa área e essa seção visa a apresentar uma síntese da situação dessa área de pesquisa, descrevendo os principais trabalhos desenvolvidos e formas de agrupá-los.

Tanto a geração automática de dados de teste quanto a automatização de oráculos de teste sofrem com as limitações inerentes da própria atividade de teste (HOWDEN, 1987a; RAPPS; WEYUKER, 1985; HARROLD, 2000). Em geral, os seguintes problemas são indecidíveis do ponto de vista computacional:

Correção: não existe um algoritmo de propósito geral que prove a correção de um produto de software;

Equivalência: dados dois programas, não existe um algoritmo de propósito geral capaz de dizer se eles são equivalentes; ou dados dois caminhos (sequências de comandos) identificar se computam a mesma função;

Executabilidade: dado um caminho qualquer (sequência de comandos) não existe um algoritmo de propósito geral capaz de encontrar um valor de entrada que cause a execução desse caminho;

Caminho ausente: corresponde a uma determinada funcionalidade requerida para o programa, mas que por engano não foi implementado, isto é, o caminho correspondente não existe no programa; e

Correção coincidente: a existência de dois ou mais defeitos pode fazer um produto, coincidentemente, apresentar um resultado correto, pois um defeito pode mascarar o outro.

Essas limitações, descritas e exemplificadas mais detalhadamente em Vergilio, Maldonado e Jino (2007), impedem uma completa automação de todas as tarefas exigidas durante a atividade de teste.

Quando se testa um produto de software, o que se deseja avaliar é sua estrutura ou as funções que este implementa. Como visto anteriormente, no primeiro caso, o teste é conhecido como teste estrutural ou caixa-branca e o segundo como teste funcional ou caixa-preta. Independentemente da técnica de teste utilizada dados de teste são escolhidos para exercitar (executar) o programa em teste e, durante essa execução, a correção,

desempenho e/ou a qualidade do programa podem ser avaliados. Dependente do domínio de aplicação, conjuntos de teste podem estar disponíveis mas, em geral, novos dados de teste devem ser gerados constantemente. A atividade de geração de dados de teste é bastante complexa e envolve muitos passos e cada passo tem uma série de questões relacionadas. A Figura 2.3, adaptada de Edvardsson (1999), mostra a arquitetura básica de um GDT, considerando uma abordagem de geração orientada a caminho.

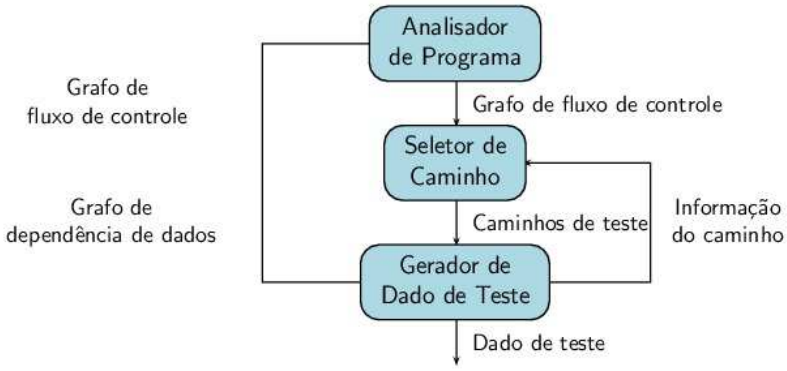


Figura 2.3: *Arquitetura de um GDT Orientado a Caminho.*

O gerador da Figura 2.3 analisa o programa em teste por meio do grafo de fluxo de controle ou do grafo de dependência de dados. No passo seguinte, o seletor de caminhos identifica um conjunto de caminhos para satisfazer determinado critério de teste e, no último passo, os dados de teste são gerados para satisfazer os caminhos identificados. É importante observar que a geração de dados de teste não é gerada em um único passo, mas, sim, em vários. E um conjunto de teste adequado não pode ser obtido se qualquer um desses passos for realizado de forma incorreta.

Em termos de pesquisa, diferentes representações para distinguir as técnicas de geração automática de dados de teste (GADT) são possíveis, tais como, baseadas em especificação, baseada em código, aleatória, estática, dinâmica, dentre outras. A Figura 2.4, adaptada de Mahmood (2007), destaca as áreas que técnicas de GADT podem pertencer. Aqui também pode ser observado que existe uma complementariedade entre as técnicas funcional e estrutural principalmente pelo fato que, em geral, elas são utilizadas em momentos diferentes no teste dos produtos de software.

Conforme ilustrado na Figura 2.4, as técnicas da GADT estão divididas em duas partes, ou seja, teste funcional ou teste estrutural que foram definidas nas Seções 2.4.1 e 2.4.2, respectivamente. A seguir, as subdivisões definidas por Mahmood (2007) são apresentadas.

A abordagem de GDT **Aleatório** consiste simplesmente na geração de entradas ao acaso (KOREL, 1996). Essa abordagem é rápida e simples, mas pode não ser uma boa escolha para sistemas complexos ou se utilizada em conjunto com critérios de adequação

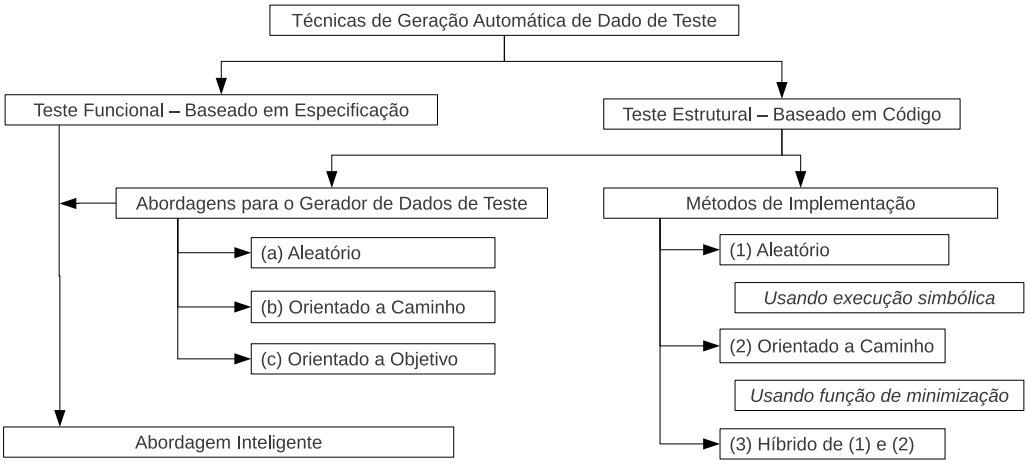


Figura 2.4: *Revisão das Técnicas de GADT (adaptada de Mahmood (2007)).*

complexos. A probabilidade de selecionar uma entrada adequada ao acaso pode ser muito baixa.

Já a abordagem de GDT **Orientada a Caminho** trata-se de uma abordagem típica de geração de caminhos a partir do grafo de fluxo de controle do programa. Nessa abordagem, inicialmente o grafo é criado e, posteriormente, um caminho particular é selecionado. Auxiliado por uma técnica de avaliação simbólica, considerando análise estática, ou função de minimização, considerando uma análise dinâmica, o dado de teste que executa o caminho pode ser gerado. No caso de execução simbólica, variáveis são usadas ao invés de valores reais durante a travessia no caminho (HAJNAL; FORGáCS, 1998; PARGAS; HARROLD; PECK, 1999). A existência de caminhos não executáveis é um problema que dificulta a geração de dados de teste para qualquer caminho selecionado, além da complexidade dos tipos de dados empregados no programa em teste.

Para a abordagem de GDT **Orientada a Objetivo**, um dado de teste é selecionado a partir de um conjunto de entrada já disponível visando à execução do objetivo desejado, tal como um comando, independentemente do caminho escolhido para atingir o objetivo (PARGAS; HARROLD; PECK, 1999). Dois passos básicos são realizados: 1) identificar o conjunto de comandos (e os arcos respectivos) que, se cobertos, implica atingir o critério; e 2) gerar o dado de entrada que executa cada comando selecionado (e os arcos respectivos) (GOTLIEB; BOTELLA; RUEHER, 1998). As abordagens "baseada em assertiva" e "encadeamento" são caracterizadas como orientadas a objetivo. Na primeira assertivas são inseridas e então resolvidas enquanto que a segunda exige que seja realizada uma análise de dependência de dados.

A abordagem **Inteligente** consiste na utilização de sofisticadas análises do código ou de sua especificação, utilizando técnicas de IA, para guiar a busca por novos

dados de teste (MICHAEL; MCGRAW, 1998; TRACEY; CLARK; MANDER, 1998; PARGAS; HARROLD; PECK, 1999).

Existem **Métodos Estáticos** que consistem naqueles utilizados para a análise e verificação de representações do sistema, tais como o documento de requisitos, diagramas de projeto e o código fonte do software, de forma manual ou automática sem exigir sua execução (CHU; DOBSON; LIU, 1997; SOMMERVILLE, 2007). Em geral, a análise estática é realizada por meio de execução simbólica que visa a identificar as restrições das variáveis de entrada para um critério de teste particular. Soluções a essas restrições representam dados de teste (TRACEY; CLARK; MANDER, 1998). Já os **Métodos Dinâmicos**, ao invés de empregarem substituição de variáveis como na execução simbólica, executam o programa em teste com algum valor de entrada gerado, possivelmente, de forma aleatória (CHU; DOBSON; LIU, 1997). Monitorando o fluxo de execução do programa, é possível identificar se o caminho desejado foi ou não percorrido. Em caso negativo, retorna-se até o ponto onde o caminho foi desviado e, utilizando-se métodos de busca diferentes, as entradas podem ser manipuladas até que a direção correta seja tomada.

Dentro desse contexto, os **Métodos híbridos** combinam métodos estáticos e dinâmicos de modo que os benefícios de cada método possam ser utilizados de forma sincronizada, visando a facilitar a geração do dado de teste desejado. Por exemplo, no trabalho de Meudec (2001), a execução simbólica é utilizada pela abordagem dinâmica.

2.5.1 Classificação das Técnicas de Geração de Dados de Teste

Mahmood (2007) conduziu uma revisão sistemática sobre o tema de geração automática de dados de teste. A Tabela 2.1 sintetiza os dados coletados e os classifica em termos da categoria do teste apoiado, abordagem de geração utilizada e método de implementação.

Como observado por Mahmood (MAHMOOD, 2007), no período em que a revisão foi realizada (1997 a 2006), sempre houve produção anual sobre o tema de pesquisa, embora a quantidade tenha variado de ano para ano. A maioria dos artigos são de pesquisas colaborativas com o esforço de dois ou mais pesquisadores, o que pode indicar o grau de dificuldade da área. Existem soluções que atendem às técnicas de teste caixa-preta (TCP), caixa-branca (TCB) e caixa-cinza (TCC), que utiliza as técnicas de caixa-preta e de caixa-branca juntas, e utilizam as várias abordagens: aleatória (Ale.), orientada a caminho (Cam.) e orientada a objetivo (Obj.), sendo que a forma de implementação por métodos dinâmicos (Din.) é a mais utilizada, seguido do método híbrido (Híb.) e por último estático (Est.). A abordagem estática é mais utilizada para geração de dados de teste manuais ou semiautomáticas.

Tabela 2.1: *Estatística de Pesquisa e Tendências dos Artigos Sobre GADT (adaptada de Mahmood (2007)).*

Referência	Ano de publicação	Categoria de Teste			Abordagem			Implementação		
		TCP	TCB	TCC	Ale.	Cam.	Obj.	Est.	Din.	Híb.
(KOREL, 1990)	1990		X			X			X	
(CHU; DOBSON; LIU, 1997)	1997	X			X					X
(GALLAGHER; NARASIMHAN, 1997)	1997	X				X			X	
(MICHAEL et al., 1997)	1997	X			X				X	
(OFFUTT; PAN, 1997)	1997		X			X		X		
(GOTLIEB; BOTELLA; RUEHER, 1998)	1998		X				X			X
(GUPTA; MATHUR; SOFFA, 1998)	1998		X		X				X	
(HAJNAL; FORGáCS, 1998)	1998			X		X				X
(KOREL; AL-YAMI, 1998)	1998		X				X		X	
(MICHAEL; MCGRAW, 1998)	1998	X			X				X	
(TRACEY; CLARK; MANDER, 1998)	1998	X	X			X		X	X	
(TRACEY et al., 1998)	1998			X	X				X	
(YANG; SOUTER; POLLOCK, 1998)	1998		X			X		X		
(BOUSQUET; ZUANON, 1999)	1999	X			X				X	
(GUPTA; MATHUR; SOFFA, 1999)	1999		X			X			X	
(HOFFMAN; STROOPER; WHITE, 1999)	1999	X			X				X	
(JENG; FORGáCS, 1999)	1999	X			X					X
(OFFUT; LIU, 1999)	1999	X			X				X	
(PARGAS; HARROLD; PECK, 1999)	1999		X				X		X	
(BUENO; JINO, 2000)	2000		X			X			X	
(SOFFA; MATHUR; GUPTA, 2000)	2000		X			X			X	
(TAYLOR; CUKIC, 2000)	2000		X			X			X	
(EDWARDS, 2001)	2001	X				X			X	
(EDVARDSSON; KAMKAR, 2001)	2001		X			X				X
(GOURAUD et al., 2001)	2001		X		X	X			X	
(MEUDEC, 2001)	2001		X			X			X	
(LIN; YEH, 2001)	2001		X				X		X	
(MICHAEL; MCGRAW; SCHATZ, 2001)	2001	X			X				X	
(SY; DEVILLE, 2001)	2001		X			X			X	
(VISVANATHAN; GUPTA, 2002)	2002		X			X			X	
(DÍAZ; TUYA; BLANCO, 2003)	2003		X				X			X
(EMER; VERGILIO, 2003)	2003		X			X			X	
(SY; DEVILLE, 2003)	2003		X			X			X	
(BARESEL et al., 2004)	2004		X				X		X	
(KHOR; GROGONO, 2004)	2004		X			X			X	
(MANSOUR; SALAME, 2004)	2004		X			X			X	
(LIU et al., 2005)	2005			X	X				X	
(ALSHRAIDEH; BOTTACI, 2006)	2006		X			X			X	
(BOTELLA; GOTLIEB; MICHEL, 2006)	2006		X			X		X	X	
(GALLAGHER; OFFUTT; CINCOTTA, 2006)	2006		X			X			X	
(MCMINN et al., 2006)	2006		X				X			X
Total		11	28	3	13	23	7	4	32	7

Para ilustrar o processo de criação de um dado de teste, foi escolhida a abordagem proposta por Korel (1990), por ser um dos representantes das abordagens com mais demandas de pesquisa, conforme Tabela 2.1, ou seja, apoia a geração de dados de teste para critério caixa-branca, utilizando uma abordagem orientada a caminho e que emprega métodos dinâmicos.

Utilizando o exemplo clássico da classificação de triângulos (MCMINN, 2004), a listagem da Figura 2.5 apresenta uma possível implementação desse programa em linguagem C e ao lado está o grafo de fluxo de controle da função `tri_type`. Nesse grafo, o nó *s* (*start node*) representa o nó de entrada e o nó *e* (*exit node*) o nó de saída. O rótulo dos demais nós faz referência aos comandos que contém.

Na abordagem de Korel (1990), o procedimento de geração de dado de teste é executado em uma versão instrumentada do programa original. Buscando executar um determinado caminho do programa, uma entrada arbitrária é escolhida. Se, durante a execução, um desvio (ramo) indesejado é seguido – um que desvie do caminho desejado –, uma busca local por entradas do programa é realizada, utilizando uma função objetivo derivada a partir do predicado de interesse referente ao ramo alternativo. A função objetivo descreve quão próximo se está do predicado desejado ser verdadeiro.

```
1  /*  s  */ int tri_type(int a, int b, int c)
2             {
3               int type;
4  /*  1  */   if (a > b)
5  /* 2-4 */   {  int t = a; a = b; b = t;  }
6  /*  5  */   if (a > c)
7  /* 6-8 */   {  int t = a; a = c; c = t;  }
8  /*  9  */   if (b > c)
9  /*10-12*/   {  int t = b; b = c; c = t;  }
10 /* 13  */   if (a + b <= c)
11            {
12 /* 14  */     type = NOT_A_TRIANGLE;
13            }
14            else
15            {
16 /* 15  */     type = SCALENE;
17 /* 16  */     if (a == b && b == c)
18            {
19 /* 17  */       type = EQUILATERAL;
20            }
21 /* 18  */     else if (a == b || b == c)
22            {
23 /* 19  */       type = ISOSCELES;
24            }
25            }
26 /*  e  */   return type;
27            }
```

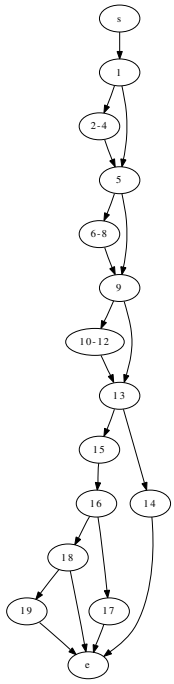


Figura 2.5: *Programa de Classificação de Triângulos (extraída de McMinn (2004)).*

Por exemplo, considere que se deseja executar o caminho $< s, 1, 5, 9, 10, 11, 12, 13, 14, e >$. Se a função `tri_type` for executada com a entrada $(a = 10, b = 20, c = 30)$, o fluxo de controle segue com sucesso os ramos falsos entre os nós de 1 a 5. Entretanto, o fluxo de controle diverge do desejado no nó 9. Nesse ponto, a busca local é iniciada visando a alterar os dados de entrada de modo que o ramo desejado seja executado. Assumindo que os predicados dos ramos são na forma *a op b*, sendo a e b expressões aritméticas e *op* um operador relacional, uma função objetivo na forma f *rel* 0 é derivada, sendo f e *rel* definidos conforme Tabela 2.2.

Tabela 2.2: *Funções Objetivo – Predicados Relacionais (extraída de Korel (1990)).*

Predicado relacional	f	*rel*
$a > b$	$b - a$	$<$
$a \geq b$	$b - a$	$\leq$
$a < b$	$a - b$	$<$
$a \leq b$	$a - b$	$\leq$
$a = b$	$abs\,(a - b)$	$=$
$a \neq b$	$-abs\,(a - b)$	$<$

A função f deve ser minimizada para um valor positivo (ou zero, se *rel* é $<$) quando o predicado do ramo atual para o ramo desejado for falso, ou para um valor negativo (ou zero, se *rel* é $=$ ou $\leq$) quando for verdadeiro. Considerando o predicado para o ramo verdadeiro a partir do nó 9, a função objetivo é $c - b > 0$. O valor dessa função para a entrada $(a = 10, b = 20, c = 30)$ é $30 - 20 = 10$. Desse modo, o programa

deve ser instrumentado de modo que tais valores possam ser computados. Isso pode ser feito por meio de uma expressão de ramo, como no exemplo a seguir:

```
if (eval_obj(9, b, c))
{
    ...
}
```

A função *eval_obj* reporta a distância até o nó 9, usando as variáveis b e c. A função irá retornar um valor verdadeiro correspondente á avaliação da expressão original de modo a não alterar a semântica do programa. Segundo McMinn (2004), esse mecanismo de busca local para derivar valores de entrada, utilizando uma função objetivo, recebe o nome de método de alternância de variáveis (*alternating variable method*). O valor de cada variável de entrada é ajustado individualmente, mantendo o valor das demais variáveis constantes. O primeiro estágio de manipulação das variáveis é chamado de fase exploratória. Ele investiga a vizinhança da variável, incrementando ou decrementando seu valor original, reavalia a função objetivo para identificar qual padrão causa uma melhoria na busca pelo objetivo desejado e leva à próxima fase, denominada fase padrão. Nessa fase, uma série de alterações maiores no valor da variável é realizada visando a encontrar o mínimo para a função objetivo. Atingido esse objetivo, a próxima variável é então selecionada e a fase exploratória é reiniciada.

Retomando o exemplo apresentado em que a execução tomou um rumo diferente daquele desejado a partir do nó 9, incrementar ou decrementar o valor da variável a nesse caso não altera o valor da função objetivo, de modo que essa variável tem o seu valor original mantido. Em seguida, a variável b é analisada e um decremento na variável b piora o resultado da função objetivo, mas um incremento leva a uma melhoria. Assim sendo, a fase padrão tem início e a variável b é incrementada até $b > c$. Supondo que o valor 31 é atingido, o novo vetor de entrada ficaria $(a = 10, b = 31, c = 30)$. A partir dessa nova entrada, a execução segue corretamente a partir do nó 9 como era desejado, entretanto, ela diverge novamente no nó 13 uma vez que o valor de $a + b$ nesse nó é maior que o valor de c. Uma busca local é iniciada novamente par ajustar o valor das variáveis de modo que o ramo verdadeiro seja seguido, mantendo a execução correta dos ramos anteriores. A função objetivo derivada do predicado do ramo verdadeiro é $(a+b)-c \leq 0$. Um decremento no valor da variável b causa uma violação do sub-caminho a partir do nó 9, já um incremento causa uma melhoria na função objetivo, uma vez que os valores de b e c são trocados nos nós de 10 a 12. Eventualmente, o vetor de entrada $(a = 10, b = 40, c = 30)$ é encontrado e o caminho completo desejado é executado.

É importante observar que como todo método de busca local, o resultado final é dependente da solução inicial fornecida e, dependendo desse vetor de entrada inicial a solução pode não ser encontrada em função das violações que podem ocorrer. Para mais

informações sobre métodos de busca e outros exemplos de geração de dados de teste por meio dessa abordagem, o leitor pode consultar o trabalho de McMinn (2004).

Perante o exemplo apresentado, pode-se dizer que a IA corresponde a uma linha de pesquisa situada no contexto da Ciência da Computação e tem como objetivo desenvolver, avaliar e aplicar técnicas na criação de sistemas inteligentes. Seu objetivo principal é capacitar o computador a executar funções que são desempenhadas pelo ser humano usando o conhecimento e o raciocínio (REZENDE; PRATI, 2005).

Essas aplicações têm culminado na criação de uma nova e promissora área de pesquisa na computação chamada *Search-based Software Engineering (SBSE)*, do português Engenharia de Software Baseada em Busca (ESBB), que trata da aplicação de técnicas de otimização matemática para a resolução de problemas complexos da área de Engenharia de Software. Segundo o site SEBASE (HARMAN, 2006), que mantém uma base de dados atualizada sobre trabalhos na área de SBSE, do Departamento de Ciência da Computação da Universidade College London, 52% das publicações de SBSE se concentram nas atividades de teste e depuração. Isso se deve ao alto custo de aplicação dessas atividades que, em geral, pode passar de 50% do custo de desenvolvimento (KRACHT; PETROVIC; WALCOTT-JUSTICE, 2014; DELAHAYE; BOUSQUET, 2015). Perante esse cenário foi criada uma subárea da SBSE chamada de *Search-Based Software Testing (SBST)*, que foca a aplicação de técnicas de otimização matemática na resolução de problemas no contexto da atividade de teste. Por isso, o desafio é automatizar o processo de teste, tanto quanto possível, e a geração de dados de teste é naturalmente uma parte fundamental desta automação. Dentro desse contexto, na Seção 2.5.2 são apresentadas algumas meta-heurísticas que podem ser aplicadas na atividade de teste de software.

2.5.2 Algoritmos de Busca e Meta-heurística

Uma possível estratégia que atraiu grande interesse na automação da geração de dados de teste é a aplicação e adaptação de algoritmos de busca e otimização (OSMAN; KELLY, 1996). A principal razão para tal interesse é que os problemas de geração de dados de teste muitas vezes podem ser representados como problemas de otimização.

São chamados de problemas de otimização os problemas que requerem que seus dados sejam modificados para que atinjam um certo grau de qualidade. Vão desde problemas de rota (por exemplo, o problema do caixeiro viajante) até problemas complexos de otimização de funções. Segundo Russell e Norvig (2003), os problemas podem ser definidos por quatro componentes:

1. O estado inicial – uma possível solução pertencente ao seu domínio que dará origem a busca pela melhor solução;

2. As ações possíveis – obtenção de conhecimento a fim de adequar a solução;
3. O teste de objetivo – determina se a solução encontrada é ótima ou não; e
4. Uma função objetivo – avalia quão próximo de uma solução ótima está a solução atual.

Para resolver os problemas, os algoritmos de otimização realizam uma busca de soluções em um espaço de estados por meio de uma função objetivo, também chamada de função de *fitness*, que define, matematicamente, o quão próximo a atual solução está da solução ótima, que corresponde a um máximo ou mínimo global da função. O máximo global (ou mínimo global) é a melhor solução em todo espaço de busca, o objetivo de todos os problemas de otimização. Máximo local (ou mínimo local) é a melhor solução em um sub-espaço de busca. Algoritmos de busca que melhoram seus resultados sem guardar estados anteriores tendem a ficarem presos em máximos locais.

Os algoritmos de busca podem ser utilizados quando não há um conhecimento sobre a solução real para o problema, entretanto, a função de *fitness* deve ser capaz de identificar quando a solução foi encontrada.

Essas resoluções automáticas são realizadas por meio de métodos inteligentes, nos quais a aquisição de conhecimento pode ocorrer durante a execução ou previamente. Existem diversas técnicas, sendo que cada uma possui características que as tornam aptas a resolver problemas específicos. Contudo, não existe um método capaz de resolver qualquer problema, esse deve ser escolhido de acordo com o domínio. Por isso, pesquisas são realizadas nesta área e várias novas técnicas foram propostas, cada uma aplicável a um determinado domínio. Dentre essas técnicas destacam-se:

Busca por força-bruta: também conhecida como busca cega e é considerada a mais simples. Utiliza um método no qual precisa percorrer todo o espaço de busca, que, dependendo do problema, pode demandar um esforço imenso. Toda possível solução deve ser analisada e nenhum conhecimento é adquirido durante este processo. O uso de algoritmos baseados em heurísticas pode reduzir o tempo gasto nesta operação. Entretanto, o algoritmo de força-bruta é capaz de garantir a existência, ou a ausência, de soluções para o problema, apesar de o tempo aumentar consideravelmente quando o número de estados aumenta. Mas, eventualmente, a melhor solução será encontrada. Quando não há uma limitação de tempo, ela torna-se a melhor escolha para um problema de otimização.

Otimização da Colônia de Formigas (ACO - *Ant Colony Optimization*): como o próprio nome indica, foram inspirados nas formigas, principalmente no comportamento que elas apresentam na busca por alimento, mas também no que diz respeito à organização do trabalho e cooperação entre si. Uma colônia de insetos é muito organizada e as atividades coletivas dos insetos são realizadas com a auto-organização. Detectou-se que

uma forma de comunicação entre as formigas, ou entre as formigas e o ambiente, baseia-se no feromônio, um elemento químico produzido pelas formigas.

A ideia é que as formigas movem-se randomicamente em busca de alimento, ou seja, realizam buscas exploratórias por possíveis soluções. Ao encontrarem alimento elas retornam para o ninho, depositando feromônio. A quantidade maior de feromônio significa que mais formigas encontraram este caminho e depositaram o feromônio, aumentando a probabilidade de este ser o melhor caminho ou o mais curto. Assim, este caminho tornou-se uma solução que foi otimizada em função do nível de feromônio encontrado na trilha. O algoritmo de colônia de formigas foi criado por Marco Dorigo em 1992 (DORIGO; GAMBARDELLA, 1997).

Enxame de Partículas (PSO - *Particle Swarm Optimization*): foi apresentado em 1995 como uma técnica que se destacou pela sua simplicidade, robustez e eficiência (KENNEDY; EBERHART, 1995). O seu desenvolvimento se baseia no comportamento coletivo de animais que vivem em sociedade, tais como enxame de abelhas, bando de pássaros e cardume de peixes. Cada membro deste enxame é movimentado dentro do espaço de busca do problema por duas forças. Uma os atrai com uma magnitude aleatória para a melhor localização já encontrada por ele próprio e outra para a melhor localização encontrada entre alguns ou todos os membros do enxame. A posição e a velocidade de cada partícula são atualizadas a cada iteração até todo o enxame convergir. A PSO apresenta as seguintes vantagens: (1) simplicidade de implementação; (2) existência de poucos parâmetros a serem ajustados; (3) utilização de uma população relativamente pequena; e (4) necessidade de um número relativamente pequeno de avaliações da função objetivo para convergir

Subida da encosta (*Hill-Climbing*): algoritmo iterativo de melhoria, em outras palavras, ele melhora sua solução inicial passo a passo. A solução inicial é escolhida aleatoriamente e, então, é melhorada até atingir um resultado aceitável. Esta técnica não armazena estados anteriores, move-se no espaço de busca enxergando apenas o estado atual (RUSSELL; NORVIG, 2003). Portanto, não é capaz de atingir resultados ótimos, apesar de eventualmente poder atingir, mas fornece soluções aceitáveis, as quais necessitariam de um longo período de tempo para serem encontradas pelo algoritmo de força bruta. Devido à sua inicialização estocástica, este algoritmo torna-se sensível à sua inicialização, ou seja, depende fortemente de um bom estado inicial para encontrar o máximo global.

Têmpera Simulada (*Simulated Annealing*): o algoritmo de subida da encosta não é capaz de realizar movimentos de descida da encosta para que possa sair de um máximo local. Introduzir uma certa aleatoriedade à subida da encosta pode trazer melhorias ao seu desempenho, permitindo que este saia de picos e encontre novos máximos locais, podendo eventualmente encontrar o máximo global. Criado por Kirkpatrick, Gelatt e Vec-

chi (1983), foi inspirado na metalurgia, a têmpera é o processo utilizado para temperar ou endurecer metais e vidros, aquecendo-os a altas temperaturas e depois resfriando-os gradualmente. Um comportamento análogo é reproduzido no algoritmo da têmpera simulada, que inicia sua busca com um grau de exploração do espaço de estados alto e gradativamente reduz essa exploração.

Algoritmos Evolutivos: é uma família de algoritmos baseados em comportamentos da natureza que tem como princípio básico a evolução biológica. Dentre estes algoritmos, destaca-se o algoritmo genético (AG) (GOLDBERG, 1989). Um algoritmo genético é uma técnica de busca utilizada na ciência da computação para achar soluções aproximadas em problemas de otimização e busca e são uma classe particular de algoritmos evolutivos que usam técnicas inspiradas pela biologia evolutiva como hereditariedade, mutação, seleção natural e recombinação (ou *crossing over*) (GOLDBERG, 1989).

Segundo Harman, Mansouri e Zhang (2012), dentre as meta-heurísticas descritas, os AGs se destacam como a técnica de otimização mais empregada nas publicações desde 1976 a 2008. Isso também se confirma com o número de trabalhos encontrados na literatura com a aplicação do mapeamento sistemático descrito no Capítulo 4. Devido a isso e ao uso dessa meta-heurística neste trabalho, ela é detalhada no Capítulo 5, junto com o AG proposto nesta tese.

2.6 Considerações Finais

Neste capítulo foi apresentada uma introdução à atividade de teste de software. Inicialmente os fundamentos do teste de software foram discutidos e em seguida foram apresentados alguns conceitos básicos para o entendimento das fases, técnicas e critérios de teste, além da definição das terminologias de teste de software utilizadas ao longo deste trabalho. Foram apresentadas a técnica de teste funcional, com os critérios de particionamento em classes de equivalência, análise do valor limite e grafo causa-efeito, a técnica teste estrutural, com os critérios baseados em complexidade, fluxo de controle e fluxo de dados e a técnica baseada em defeitos, com os critérios de semeadura de defeitos e teste de mutação. Além disso, foi abordado o conceito de geração de automática de dados de teste descrevendo algumas técnicas e meta-heurísticas aplicadas nesse contexto. Este capítulo serve como fundamentação teórica para a proposta de trabalho apresentada nesta tese.

CAPÍTULO 3

Teste WUI Baseado em Modelos

3.1 Considerações Iniciais

Modelar na engenharia de software é o processo de criação de uma representação de um sistema, abstraindo e simplificando o seu comportamento ou estrutura, ou seja, é uma forma de representar a estrutura/comportamento de um sistema. Os modelos são mais simples do que o sistema que descrevem e por isso ajudam a mais facilmente o entender.

O Teste Baseado em Modelos, MBT – *Model-Based Testing*, refere-se a um processo de engenharia de software que estuda, constrói, analisa e aplica modelos bem definidos para dar suporte nas varias atividades relacionadas com os testes. É uma técnica de teste caixa preta que tem por objetivo verificar se a implementação de um software se encontra de acordo com a suas especificações e foca-se na geração automática de testes. Estes teste servirão para verificar se a implementação se encontra de acordo com o modelo.

A ideia é identificar e construir um modelo abstrato que represente o comportamento do SUT e, com esse modelo, é possível gerar dados de teste. Porém, a quantidade de dados de teste gerada pode ser muito grande, principalmente se o modelo representar a UI da SUT. Segundo Memon (2007), grande parte dos softwares utiliza interfaces gráficas para a interação com o usuário, sendo que as interfaces modernas têm seu tamanho e complexidade cada vez maiores.

Neste capítulo são apresentados conceitos relacionados a MTB e ao teste de interfaces de software. São explanados conceitos, definições, limitações e aplicabilidade desses testes.

3.2 Terminologias e Conceitos Básicos

Em informática, as interfaces gráficas GUI e WUI permitem ao usuário interação com dispositivos digitais por meio de elementos gráficos como ícones e outros indicadores visuais, em contraste à interface de linha de comando (CLI). A interação é feita

geralmente por meio do mouse, teclado ou gestos, com os quais o usuário é capaz de selecionar símbolos e manipulá-los de forma a obter algum resultado prático. Esses símbolos são designados, em alguns trabalhos da literatura, de *widgets*. Nesse texto, será utilizado o termo objeto como sinônimo de *widget*.

Sabe-se que até 60% do código-fonte destas aplicações costuma ser destinado à construção da interface e o tratamento da sua lógica (MEMON; POLLACK; SOFFA, 2001a). Por este motivo, observa-se que não é suficiente aplicar métodos tradicionais para testar apenas a lógica de negócios no nível das classes ou módulos, como o proposto por Beck (2002) e chamado de *Test-driven development*. É necessário testar também o comportamento da interface e estudar a relação desses testes. Porém, módulos que implementam a lógica de interação humana são mais difíceis de testar quando comparados com módulos que implementam puramente a lógica de negócios. As principais dificuldades encontradas para testar estes módulos são:

1. A comunicação entre a interface e os componentes que implementam o comportamento é realizada por meio de sinais oriundos de dispositivos físicos como mouse e teclado, sendo que estes são difíceis de simular;
2. Os casos de teste devem explorar as inúmeras combinações de elementos da interface, o que torna o processo custoso;
3. Características visuais como o leiaute de interface não devem afetar o teste; e
4. A cobertura de testes não deve se basear apenas nas linhas de código exercitadas, e sim no número de diferentes combinações de utilização de componentes de interface gráfica. Ou será que existe uma relação direta entre elas?

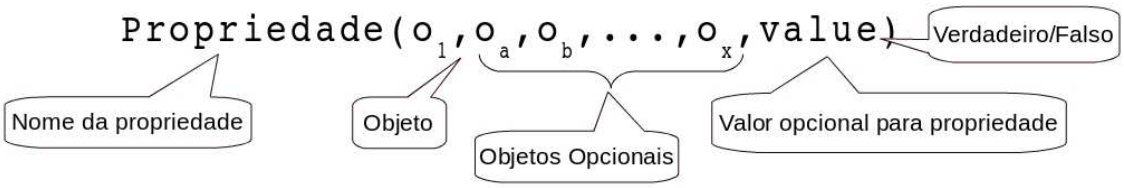
Essas interfaces utilizam um ou mais metáforas para objetos familiares na vida real, tais como botões, menus, uma área de trabalho ou até mesmo a estrutura de uma sala de trabalho. Objetos de uma GUI incluem elementos tais como janelas, menus suspensos, botões, barras de rolagem e imagens. O usuário de software realiza eventos para interagir com a interface gráfica e manipulação desses elementos.

De uma forma geral, em interfaces gráficas, um estado é modelado como um conjunto de objetos (*label*, *form*, *button*, *text*, etc.) e um conjunto de propriedades desses objetos (*background-color*, *font*, *caption*, etc.). Cada interface usará certos tipos de objetos com propriedades associadas; em qualquer ponto no tempo específico, a interface pode ser descrita em termos dos objetos que contém e os valores das suas propriedades.

Formalmente, uma interface é modelada em determinado momento t em termos:

- Seus objetos $O = \{o_1, o_2, ..., o_m\}$; e
- As propriedades $P = \{p_1, p_2, ..., p_l\}$ desses objetos. Cada propriedade p_i é uma relação booleana n_i-ary, para $n_i \geq 1$, onde o primeiro argumento é um objeto

$o_1 \in O$. Se $n_i > 1$, o último argumento pode ser tanto um objeto ou um valor de propriedade, e todos os argumentos intermediários são objetos. A Figura 3.1(a) mostra a estrutura de propriedades. O valor da propriedade (opcional) é uma constante escolhida de um conjunto associado com a propriedade em questão: por exemplo, a propriedade "$background - color$" tem associado um conjunto de valores, $\{white, yellow, pink, etc.\}$. Um conjunto distinto de propriedades, o *object types*, que são relações unária, ("*windows*","*button*") é assumida para estar disponível. A Figura 3.1(b) ilustra um objeto de botão chamado *Button*1. Uma de suas propriedades é chamado *Caption* e seu valor atual é "*Entrar*".



(a) *Estrutura de Propriedades.*

(b) *Objeto Botão com Propriedade Associada.*

Figura 3.1: *Propriedades de um Objeto.*

Um ponto que deve ser observado sobre a descrição das propriedades é que as propriedades são as relações e não funções. Assim, não podem assumir, ao mesmo tempo, vários valores para a mesma propriedade de um determinado objeto.

Para se criarem modelos de interfaces gráficas, os objetos e suas propriedades devem ser identificados. Segundo Memon (2001), na prática, existem algumas maneiras (abordagens) de se fazer isso:

1. Examinando manualmente – o testador examina as interfaces e identifica todos os objetos e suas respectivas propriedades. Esta abordagem é propensa à incompletude, especialmente porque as interfaces podem ter propriedades ocultas que devem ser verificadas, mas não são visíveis. Por exemplo, a ordem em que os objetos recebem $focus$ quando a tecla *Tab* é pressionada;
2. Examinando as especificações – As propriedades e tipos de objetos são extraídos das especificações das interfaces que as descrevem diretamente ou implicitamente. Esta abordagem proporciona um conjunto mais preciso de propriedades e tipos de objeto do que o primeiro. No entanto, as propriedades adicionais podem ter sido inadvertidamente introduzida pela plataforma de implementação, o qual, se não for testado, pode provocar efeitos indesejáveis durante a execução.

3. Examinando o conjunto de ferramentas/linguagem utilizado para o desenvolvimento das interfaces – Examinando as ferramentas e a linguagem de programação todos os seus tipos de objetos e propriedades identificados. Por exemplo, se a interface foi desenvolvida utilizando a linguagem JAVA, então os objetos seriam instâncias dos componentes do pacote *Swing* e as propriedades que correspondem às variáveis de instância de cada objeto.

A terceira abordagem pode levar a um conjunto maior de tipos de objetos e propriedades do que o segundo. Isso se justifica porque o conjunto de tipos de objetos e propriedades disponibilizados por uma linguagem ou ferramenta não podem ser usados na construção de uma interface gráfica particular. Por exemplo, pode-se usar *C++ Builder* da Borland (EMBARCADERO, 2014) para a construção de uma interface gráfica simples em que o usuário não tem permissão para manipular a cor do texto, ou seja, em que a cor do texto não influencia na execução de qualquer outro evento. Assim, há dois conjuntos de propriedades: o conjunto completo de propriedades para uma interface, que pode ser obtido utilizando-se a terceira abordagem, e o conjunto reduzido, que inclui apenas aquelas que seriam identificadas pela segunda abordagem, baseando-se nas especificações. Note-se que o conjunto reduzido é sempre um subconjuntos do conjunto completo de propriedades (MEMON, 2001).

Uma descrição de estado da interface está relacionada com seus objetos e suas respectivas propriedades. Esse estado contém informação sobre os tipos de todos os objetos atualmente existentes na interface, bem como todas as propriedades de cada um desses objetos. Com isso, as características importantes de UI incluem sua orientação gráfica, eventos de entrada, estrutura hierárquica, os objetos que contêm e atribuições as propriedades desses objetos. Esse conceito é formalizado na Definição 1.

Definição 1 (Estado UI) *O estado de uma interface gráfica em um determinado momento t é o conjunto P de todas as propriedades de todos os objetos O que a interface contém.*

A seguir são formalizados os conceitos sobre WUI necessários para criação dos modelos utilizados para geração de dados de teste.

3.3 Formalização de Conceitos WUI

Como explicado anteriormente, a interface do usuário é denominada de forma diferente, dependendo do tipo de aplicação e da plataforma. No contexto deste trabalho caracterizam-se dois tipo de interfaces do usuário, a GUI para programas *desktops* e a WUI para aplicações web, tendo a primeira como definição:

Definição 2 (GUI) *Uma GUI é representada de forma gráfica para um sistema de software (desktop) que aceita como entrada eventos gerados pelo sistema ou pelo usuário, a partir de um conjunto fixo de eventos e produz uma saída gráfica determinista. A GUI contém objetos gráficos; cada objeto tem um conjunto fixo de propriedades. A qualquer momento durante a execução da GUI estas propriedades podem ter valores distintos, constituindo assim o estado da GUI e organizada de forma hierárquica.*

Já as WUIs são definidas como as interfaces das aplicações web. Representam a "porta de entrada" para utilização de um software, normalmente formado de vários programas, possivelmente implementados em diferentes linguagens, simultaneamente executado em várias plataformas e conectado pela Internet. O usuário interage com o WUI, por meio de um navegador web, sem o conhecimento do software subjacente, topologia utilizada ou as plataformas de implementação. O usuário WUI espera que todo o sistema funcione como se estivesse sendo executado localmente.

Semelhante a GUI, a entrada para a WUI é sob a forma de eventos, em geral, produzindo uma saída gráfica. Pode-se afirmar que as WUIs tem todas as características das GUIs, incluindo entrada orientada a eventos que muda o estado do WUI, saída gráfica, estrutura hierárquica e objetos gráficos com propriedades. Porém, as WUIs têm características especiais como sincronismo, restrições de sincronização e um grande número de requisitos de portabilidade, o que torna o seu teste ainda mais complexo e desafiador do que o teste GUI.

As principais características das WUIs incluem sua orientação gráfica, conectividade com a Internet, orientada a eventos de entrada, *frames*, páginas e as restrições entre as páginas, os objetos que eles contêm, restrições entre os objetos e propriedades e atributos desses objetos.

O teste baseado em WUI é o foco deste trabalho e tem como principal objetivo, a partir das possibilidades de eventos da interface, testar determinada aplicação. Por isso, procurou-se formalizar os conceitos necessários para criação de modelos que permitam a geração de dados de testes. Formalmente, a classe de interfaces gráficas pode ser definida como se segue:

Definição 3 (WUI) *Uma WUI é uma GUI em que a estrutura hierárquica consiste em quadros e páginas, com restrições geométricas e temporais entre as páginas. Cada página contém objetos e restrições entre os objetos. A WUI fornece uma "porta de entrada" gráfica para o usuário interagir com o software que consiste em vários programas, possivelmente implementados em linguagens diferentes, simultaneamente em execução em várias plataformas e todos conectados pela Internet, em geral em uma arquitetura cliente–servidor.*

Uma representação da WUI e suas operações deve incluir uma representação dos múltiplos programas que determinam o estado da WUI. Esses programas podem executar no servidor e produzir saída estática (tais como *HTML*) ou saída dinâmica (tais como *DHTML*) geradas em tempo real para serem exibidas na janela do navegador. Outros programas, como JAVA *Applets*, podem executar no cliente local e sua interface pode ser exibida como parte da WUI. Verificar a exatidão da WUI deve incluir a verificação das GUIs destes programas individualmente. Podem existir relações de sincronização entre esses programas, que também devem ser verificadas. A interação do usuário com a WUI pode gerar três tipos de eventos:

1. Os disponíveis no navegador – tais como recortar e copiar;
2. Aqueles disponíveis na janela do navegador – como clicar em links, selecionando um item de uma lista e clicar em botões; e
3. Aqueles fornecidos pela WUI (vários programas em execução no navegador) – tais como JAVA *Applets* e *plug-ins*.

Um estado WUI depende amplamente das condições de ambiente em que é executado. Tais condições incluem o estado do servidor, cliente e rede. Exemplos de estado do servidor incluem sua velocidade e o estado de seu sistema de arquivos. Já para o estado do cliente, inclui a resolução do monitor, configurações de segurança, instalação de componentes, localização geográfica e hardware instalado. Para a rede, inclui-se sua velocidade e conectividade. Quanto se testa uma WUI, essas "condições de ambiente" devem fazer parte das entrada de teste.

Uma WUI contém objetos concebidos para aceitar a entrada de um usuário e a perspectiva de saída a ser exibida no navegador. Exemplos de objetos incluem itens de texto, caixas de texto, imagens, JAVA *Applets*, botões e links. Esses objetos WUI são logicamente agrupados em páginas (Definição 4); e páginas, em quadros. Note-se que estes agrupamentos permitem aumentar a usabilidade do WUI, exibindo objetos relacionados juntos. Intuitivamente, a página cria um leiaute de WUI objetos para o *browser* e estabelece relações de temporização e sincronização entre eles. Formalmente, uma página é definida como se segue:

Definição 4 (Página) *Uma página é um par* (O,C)*, onde cada* $o \in O$ *é um objeto WUI e cada* $c \in C$ *é uma restrição para os elementos de O.*

Exemplos comuns de restrições são restrições geométricas que definem o leiaute dos objetos da WUI e restrições de sincronização temporal. Note-se que os níveis adicionais de agrupamento podem ser representados de forma semelhante. Por exemplo, *frames* podem ser representados por restrições sobre um conjunto de páginas. *Frames* em

WUIs forçam um diálogo pelo usuário como uma caixa de diálogo em GUIs. Eventos de dois *frames* diferentes não podem ser intercalados.

Por exemplo, a Figura 3.2 mostra uma WUI decomposta em quadros, páginas e objetos. Cada *frame* (f_1 e f_2) contém páginas (p_1, p_4 e p_5) com vários objetos ($o_1, o_2, ..., o_6$). Eventos sobre $o1, o2$ e $o3$ não podem ser intercalados com eventos sobre os objetos o_4, o_5 e o_6. Note-se que os eventos executados em o_4, o_5 e o_6 podem ser intercalados desde que as páginas p_4 e p_5 sejam exibidas no mesmo *frame* (f_2) e, por conseguinte, são visíveis simultaneamente para o utilizador. Essas características de páginas e *frames* podem ser usadas para identificar componentes WUIs semelhantes aos desenvolvidos para GUIs.

A simples WUI mostrada na Figura 3.3 pode ser modelada em termos de seus objetos (com suas propriedades) e as restrições entre eles. A WUI contém quatro objetos, $name - label$, $name - field$, $submit - button$ e $reset - button$.

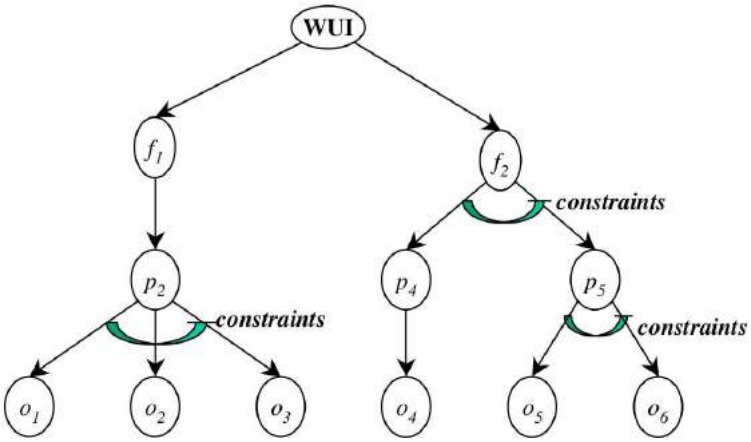


Figura 3.2: *EFG – Basico (extraída de Memon (2001)).*

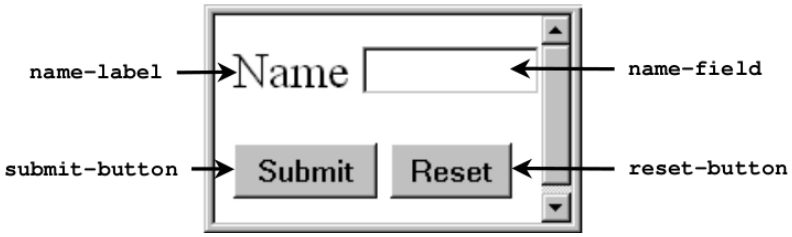


Figura 3.3: *Exemplo de WUI (extraída de Memon (2001)).*

As propriedades para cada objeto WUI descrevem as características desses objetos, como mostra a Figura 3.4 . O tipo de propriedade "*type*" (linhas 4, 6, 8 e 10) descreve o tipo do objeto, portanto, determina o seu comportamento e a interpretação das suas demais propriedades. A propriedade "*action*" (linhas 9 e 11) associa um programa

```
Frames: f1 /* A single frame */
Pages: p1 /* A single page */
Objects of p1:
  name-label: set of properties = {type("label"), value("Name"),
   color("Black"), font("Type Roman")}.
  name-field: set of properties = {type("text-field"), value(""),
   editable("TRUE")}.
  submit-button: set of properties = {type("button"), caption("Submit"),
   action("POST")}.
  reset-button: set of properties = {type("button"), caption("Reset"),
   action("RESET")}.
Constraints: /* geometric constraints imposed by the HTML code */
  {first-object(name-label), after(name-label, name-field),
   new-line(submit-button), after(submit-button, reset-button)}
```

Figura 3.4: *Propriedades da WUI (extraída de Memon (2001)).*

executável com o objeto em questão. Por exemplo, $submit - button$ e $reset - button$ tem as ações *POST* e *RESET* associadas a eles.

Note-se que a representação WUI é mais complexa do que a GUI. Em WUIs, temporização e restrições desempenham papéis importantes na sua execução. Aí surge a pergunta: Como lidar com esses desafios? A indústria de software está tentando lidar com isso aumentando o uso da automação da atividade de teste. Com isso, algumas alternativas para automação desses testes são utilizadas. As principais técnicas encontradas na literatura são: *Capture/Playback*, Programação de *Scripts*, *Data-driven* e *Keyword-driven*.

A técnica *Capture/Playback* consiste em, utilizando uma ferramenta de automação de teste, gravar as ações executadas por um usuário sobre a WUI de uma aplicação e converter estas ações em *scripts* de teste que podem ser executados quantas vezes for desejado. Cada vez que o *script* é executado, as ações gravadas são repetidas, exatamente como na execução original. Para cada caso de teste, é gravado um *script* de teste completo que inclui os dados de teste (dados de entrada e resultados esperados), o procedimento de teste (passo a passo que representa a lógica de execução) e as ações de teste sobre a aplicação. A vantagem dessa técnica é sua simplicidade de uso, sendo uma boa abordagem para testes executados poucas vezes. Entretanto, são várias as desvantagens desta técnica ao se tratar de um grande conjunto de dados de teste automatizados, tais como: alto custo e dificuldade de manutenção, baixa taxa de reutilização, curto tempo de vida e alta sensibilidade a mudanças no software a ser testado e no ambiente de teste. Como exemplo de um problema desta técnica, uma alteração na interface gráfica da aplicação poderia exigir a regravação de todos os *scripts* de teste (FEWSTER; GRAHAM, 1999; ALSMADI, 2008). Outro problema é que os *scripts* gerados a partir desse método podem conter valores *hard-coded*, ou seja, dados diretamente inseridos em um programa que não podem ser facilmente mudados e tratados.

A técnica de Programação de *Scripts* é uma extensão da técnica *capture/playback*. Por meio da programação os *scripts* de teste gravados são alterados para que desempenhem um comportamento diferente do *script* original durante sua execução. Para

que esta técnica seja utilizada, é necessário que a ferramenta de gravação de *scripts* de teste possibilite a edição. Desta forma, os *scripts* de teste alterados podem contemplar uma maior quantidade de verificações de resultados esperados automaticamente, as quais seriam realizadas pelo testador humano, normalmente, apenas de forma visual e intuitiva. Além disso, a automação de um caso de teste similar a um já gravado anteriormente pode ser feita por meio da cópia de um *script* de teste e sua alteração em pontos isolados, sem a necessidade de uma nova gravação. A programação de *scripts* de teste é uma técnica de automação que permite, em comparação com a técnica *capture/playback*, maior taxa de reutilização, maior tempo de vida, melhor manutenção e maior robustez dos *scripts* de teste. No exemplo de uma alteração na interface gráfica da aplicação, seria necessária somente a alteração das partes necessárias para refletirem as alterações sofridas pela interface. Apesar destas vantagens, sua aplicação pura também produz uma grande quantidade de *scripts* de teste, visto que para cada caso de teste deve ser programado um *script* de teste, o qual também inclui os dados de teste e o procedimento de teste. As técnicas *Data-driven* e *Keyword-driven*, que são versões mais avançadas se comparadas a essa técnica, permitindo a diminuição da quantidade de *scripts* de teste, melhorando a definição e a manutenção de casos de teste automatizados (FEWSTER; GRAHAM, 1999).

A técnica *Data-driven*, chamada de técnica orientada a dados, consiste em isolar, dos *scripts* de teste, os dados de teste, que são específicos do caso de teste, e armazená-los em arquivos separados. Os *scripts* de teste passam a conter apenas os procedimentos de teste (lógica de execução) e as ações de teste sobre a aplicação, que normalmente são genéricos para um conjunto de teste. Assim, os *scripts* de teste não mantêm os dados de teste no próprio código, obtendo-os diretamente de um arquivo separado, somente quando necessário e de acordo com o procedimento de teste implementado. A principal vantagem dessa técnica é que se pode facilmente adicionar, modificar ou remover dados de teste, ou até mesmo casos de teste inteiros, com pequena manutenção dos *scripts* de teste. Esta técnica de automação permite que o projetista de teste e o implementador de teste trabalhem em diferentes níveis de abstração, dado que o projetista de teste precisa apenas elaborar os arquivos com os dados de teste, sem se preocupar com questões técnicas da automação de teste (FEWSTER; GRAHAM, 1999).

A técnica *Keyword-driven*, também conhecida como técnica orientada a palavras-chave, consiste em extrair, dos *scripts* de teste, o procedimento de teste que representa a lógica de execução. Os *scripts* de teste passam a conter apenas as ações específicas de teste sobre a aplicação, as quais são identificadas por palavras-chave. Essas ações de teste são como funções de um programa, podendo inclusive receber parâmetros, que são ativadas pelas palavras-chave a partir da execução de diferentes casos de teste. O procedimento de teste é armazenado em um arquivo separado, na forma de um conjunto ordenado de palavras-chave e respectivos parâmetros. Assim, os *scripts* de teste não man-

têm os procedimentos de teste no próprio código, obtendo-os diretamente dos arquivos de procedimento de teste. A principal vantagem da técnica *keyword-driven* é que se pode facilmente adicionar, modificar ou remover passos de execução no procedimento de teste com necessidade mínima de manutenção dos *scripts* de teste, permitindo também que o projetista de teste e o implementador de teste trabalhem em diferentes níveis de abstração (FEWSTER; GRAHAM, 1999). A Tabela 3.1 apresenta uma síntese das técnicas descritas.

Tabela 3.1: *Síntese das Principais Técnicas de Automatização de Teste.*

Capture/Playback	Por meio de uma ferramenta de automação é realizada a gravação das ações executadas pelo usuário interagindo com a interface gráfica da aplicação. Com estas informações criam-se *scripts* de teste que podem ser executados quantas vezes forem necessários.
Programação de *Scripts* (Manuscrito)	Extensão da técnica *capture/playback*, além da gravação o *script* original pode ser editado para que desempenhe um comportamento diferente do *script* original, passando a contemplar um maior número de verificações.
Data-Driven (Orientada a Dados)	Nesta técnica é feita a extração dos dados de teste especifico a cada script e os armazenam em arquivo separado. Assim, os *scripts* de teste somente mantêm os procedimentos de teste e não os dados de teste no próprio código, obtendo-os diretamente em um arquivo separado, sendo acessado somente quando necessário e de acordo com o procedimento de teste implementado.
Keyword-Driven (Orientada a Palavra Chave)	Nesta técnica é feita a extração do procedimento de teste que representa a lógica de execução. O *script* passa a conter apenas as ações de teste sobre o software, identificadas por palavras-chave que podem trabalhar como funções de programações e receber parâmetros.

3.4 Testes Baseados em Modelos

Um modelo é uma descrição do comportamento do sistema fornecendo subsídios para compreensão e previsão do seu comportamento. O MBT é uma estratégia que envolve o desenvolvimento e utilização de um modelo que é essencialmente uma especificação de entradas para o software. Trata-se de uma técnica para a geração de casos de teste utilizando o modelo gerado como artefato de entrada (SANTOS-NETO; RESENDE; PáDUA, 2007). Surgiu como uma estratégia viável para controlar a qualidade do software, reduzindo os custos relacionados ao processo de testes, visto que os casos de teste podem ser gerados a partir de artefatos (modelos) produzidos ao longo do processo de desenvolvimento de software. A exploração desses modelos possibilita ao Engenheiro de Software a realização de uma verificação adicional dos modelos produzidos ao longo do desenvolvimento de software antes que estes sejam passados a fases seguintes do processo, uma vez que estes modelos serão utilizados para a geração dos testes e eles precisam estar corretos para garantir o sucesso desta atividade. Dessa forma, MBT contribui para a qualidade do software não apenas por meio da execução dos casos de testes após o desenvolvimento do software, mas também com a verificação dos modelos durante o processo de geração dos casos de testes. Para Santos-Neto, Resende

e Pádua (2007), o objetivo do MBT é aumentar a eficiência e qualidade dos testes de software, automatizando processos de testes, de forma a diminuir o esforço de teste e evitar atividades sujeitas a enganos cometidos pelos humanos.

Dentro desse contexto, os casos de teste são derivados totalmente ou parcialmente de um modelo que descreve algum aspecto (ex: funcionalidade, segurança, desempenho, etc.) de um software (UTTING; LEGEARD, 2007). Para sua utilização é necessário que o comportamento ou estrutura do software (ou parte deles) seja formalizado por meio de modelos com regras bem definidas (tais como métodos formais, máquinas de estado finito, diagramas UML, dentre outros).

Apesar do fato de que MBT ser confundido com Geração de Casos de Teste, é preciso ressaltar a diferença entre essas definições para facilitar o entendimento. MBT usa modelos desenvolvidos durante qualquer fase do processo de desenvolvimento do software e ampliados pela equipe de teste para gerar, automaticamente, o conjunto de casos de teste. Em contrapartida, Geração de Casos de Teste é apenas uma das tarefas que compõem o processo de testes e pode ser realizada ou não, utilizando modelos de software formalizados (SANTOS-NETO; RESENDE; PáDUA, 2007). A Figura 3.5 descreve as atividades específicas associadas a MBT, que são:

1. Construir o modelo descrevendo a estrutura/comportamento do software (uma das principais diferenças em relação às demais estratégias);
2. Geração de Casos de Teste;
   (a) Gerar entradas esperadas;
   (b) Gerar resultados ou comportamentos esperados;
3. Executar os testes;
4. Comparar os resultados obtidos com os resultados esperados; e
5. Decidir as futuras ações:
   (a) modificar o modelo;
   (b) gerar mais testes;
   (c) parar os testes; ou
   (d) estimar a confiabilidade (qualidade) do software.

A construção do modelo inicia-se com o entendimento do SUT, procurando representar as funcionalidades do sistema. Com as funcionalidades entendidas, deve-se escolher um modelo formal que melhor represente os requisitos do software em questão para então se construir o modelo do mesmo. Após a construção do modelo, aplica-se a geração dos casos de teste a partir das especificações representadas no modelo. Cada caso de teste deve possuir passos e os respectivos resultados esperados. O grau de dificuldade na automação desta etapa está associado ao modelo escolhido para representar

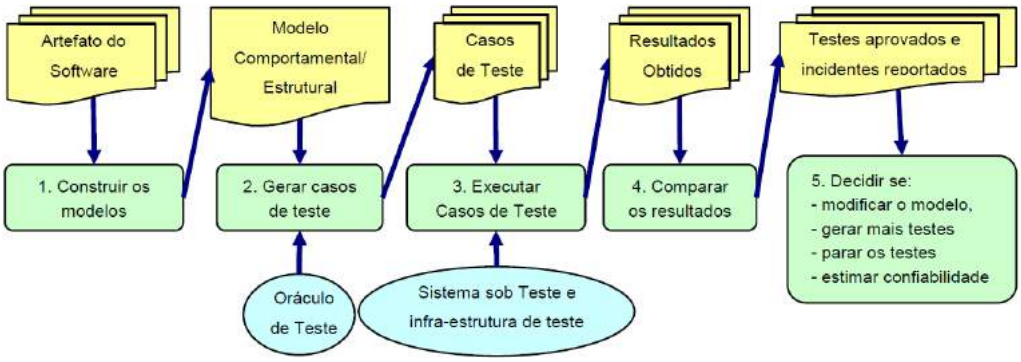


Figura 3.5: *Atividades de MBT (adaptado de Pretschner (2005)).*

o sistema sob teste. Com o conjunto de teste definido, começa-se a execução desse conjunto, esperando que a aplicação se comporte conforme o descrito nos resultados esperados. Caso isto não ocorra, é dito que o caso de teste falhou, representando a não correspondência entre implementação e especificação. Durante a execução, pode ser feita a coleta de resultados que consiste em guardar informações acerca da execução dos casos de teste, tais como tempo de execução e resultado do teste. A partir dos resultados, é possível fazer uma análise para que melhorias no modelo sejam realizadas e o processo de geração evolua no sentido de gerar casos de teste cada vez mais relevantes.

De acordo com Utting e Legeard (2007)), uma Técnica de MBT (TMBT) consiste em uma instância da estratégia de MBT apresentada na Figura 3.5 e pode ser aplicada a qualquer tipo de técnica de teste (funcional, estrutural ou baseada em defeitos). Diversas TMBTs são propostas na literatura. Elas usam diferentes modelos para especificar o software, e esses modelos normalmente descrevem diferentes características de um produto. Isso torna a identificação, seleção e utilização de uma TMBT em um projeto de software uma tarefa complexa e de difícil decisão.

As principais vantagens do MBT, segundo Neto e Travassos (2010), são:

1. Redução de custo e esforço para planejamento/execução de teste;
2. Facilidade na comunicação entre as equipes de desenvolvimento e teste;
3. Antecipação do início do processo de teste, pois o modelo pode ser construído já durante a definição dos requisitos;
4. Suporte à exposição de ambiguidade na especificação e projeto do software;
5. Capacidade de gerar e executar automaticamente testes úteis e não reduntantes;e
6. Facilidade de manutenção do conjunto de casos de teste, já que uma alteração no modelo pode ser rapidamente refletida nos casos de teste, mantendo a consistência entre eles.

Porém, sua adoção é extremamente dependente do apoio de ferramentas. Além das ferramentas, os principais desafios na adoção industrial de MBT são o conhecimento

especializado e uma quantidade considerável de esforço necessário para criar os modelos formais e o mapeamento entre o modelo e o sistema real, para se poder gerar casos de teste executáveis (GRILO; PAIVA; FARIA, 2010). Com isso, conclui-se que o esforço e conhecimentos necessários para elaboração, manualmente, dos modelos utilizados no teste baseado em modelo são obstáculos para sua adoção no meio industrial.

Quando se olha especificamente para a área de MBT, percebe-se que enquanto pesquisas recentes têm produzido resultados interessantes a respeito do desenvolvimento de novas técnicas e infra-estruturas computacionais para apoiar MBT, existe ainda uma carência por conhecimento científico sobre tais técnicas, o que dificulta na sua transferência para a indústria (GRILO; PAIVA; FARIA, 2010). Alguns fatores, identificados a partir de um estudo secundário, contribuem para este cenário:

1. Alto número de TMBTs disponível na literatura;
2. Carência de um corpo de conhecimento ou repositório com informações sobre tais técnicas; e
3. Carência de conhecimento científico (evidências) obtido a partir do uso de TMBTs em diferentes projetos de software.

Recentemente, pesquisam-se sobre teste e extração do modelo por meio de aplicações GUI. Infelizmente, a maioria dessas abordagens têm limitações e restrições sobre as aplicações GUI que podem ser modeladas, e a adoção na indústria é bem limitada.

Além dos problemas já citados, um dos principais é o fato do modelo poder estar diferente do SUT. Segundo Silva, Campos e Paiva (2008), algumas limitações do MBT devem ser pontuadas:

1. Explosão de estados – a existência de muitos estados pode fazer com que os modelos cresçam para além dos níveis administráveis. Mesmo uma simples aplicação pode conter tantos estados que a manutenção do modelo se torna difícil e de alto custo. Além disso, modelos com grande número de estados tornam a realização dos testes inviável;
2. Competências para criar modelo – o criador do modelo deve ser capaz de abstrair o estado e o comportamento do sistema e estar familiarizado com a criação de modelos (terá de entender de máquinas de estados, linguagens formais entre outros conceitos); e
3. Tempo para analisar os testes que falharam – se existiram falhas nos testes, é preciso perceber qual a causa da falha, se vem do SUT ou do modelo. O que nos testes manuais também pode acontecer, saber se a falha foi no SUT ou na *script* de teste. No entanto, como no MBT são gerados mais casos de teste e testes menos intuitivos do que os testes manuais, torna-se mais difícil e moroso encontrar a causa da falha.

São três os tipos de modelos mais utilizados para Teste GUI Baseado em Modelo (*Model-Based GUI Testing – MBGT*) e que também podem ser aplicados a WUI: o Modelo Baseado em Estados (*State-Based Model*), o Modelo Baseado em Eventos (*Event-Based Model*) e Engenharia Reversa Dinâmica.

O modelo baseado em estados é o mais utilizado (AHO et al., 2015). A ideia principal é que o comportamento de uma aplicação GUI seja representado como uma máquina de estado, nos quais os nós do modelo são chamados estados GUI, arestas são eventos e interações e cada evento de entrada pode desencadear uma transição para um estado da máquina. Um caminho é uma sequência de estados e eventos da GUI e representa um dado de teste. Existem aplicações em alguns contextos, por exemplo, a GUI *Driver* (AHO; MENZ; RATY, 2011) e *GuiTam* (MIAO; YANG, 2010a) para aplicações JAVA GUI e *AndroidRipper* (AMALFITANO et al., 2012a) para aplicativos *Android*.

Outro formato popular para extração de modelos GUI é o modelo baseado em eventos. A equipe de Atif Memon implementou a *GUITAR* (NGUYEN et al., 2014), uma ferramenta para testes automatizados GUI, por meio da construção automática de modelos com base em eventos. Memon et al. (2013) publicaram extensivamente sua pesquisa sobre GUI *Ripping*, uma técnica para extrair dinamicamente modelos baseados em eventos de aplicações GUI para fins de automação de teste. Seu objetivo é fornecer ferramentas para o processo de extração de modelo e geração de teste totalmente automatizado, mas eles não abordam o desafio de fornecer entradas específicas. Um grande desafio dessa abordagem é referente ao número de estados para modelar sistemas não triviais. Até agora, as abordagens de modelagem não forneceram soluções para este desafio (MEMON, 2007). Qualquer programa não trivial tem um grande número de estados possíveis, dependendo da definição do que é um estado e como distingui-los.

A mais recente abordagem de extração modelo de GUI é baseada em engenharia reversa dinâmica, ou seja, é feita a execução do aplicativo e observa-se o comportamento em tempo de execução da GUI. O grande desafio é passar automaticamente pela GUI fornecendo dados significativos para os campos de entrada requisitados, como usuário e senha válidos para uma tela de *login* sem instruções predefinidas do usuário (AHO; MENZ; RATY, 2011). Geralmente, alguma intervenção humana é necessária durante o processo de modelagem para alcançar uma boa cobertura com modelos de engenharia reversa dinâmica (AHO; MENZ; RATY, 2011), o que significa que a modelagem é assistida manualmente por uma pessoa durante o processo de engenharia reversa ou o modelo inicial gerado é revisado, corrigido e estendido manualmente por uma pessoa após a extração do modelo (KULL, 2012). A eficiência destas técnicas de modelagem semi-automáticas depende do grau de intervenção humana requerida (KULL, 2012). Embora existam processos semi-automáticos de fornecer valores de entrada, as abordagens mais

automatizadas não são capazes de atingir todas as partes da GUI durante a extração modelo.

Dentro desse contexto, é proposta a formalização da aplicação de MBT (UTTING; LEGEARD, 2007) dentro do contexto de WUI, que será discutido a seguir.

3.4.1 Testes WUI Baseados em Modelos

Ao testar a interface gráfica de um software, é possível detectar não só erros da interface, mas também erros relacionados com a aplicação. Dentro desse contexto, são apresentadas a seguir algumas definições adaptadas do trabalho de Memon (2001). Em seu trabalho, Memon teve como foco aplicações com interfaces gráficas do usuário para sistemas *Desktop*, ou seja, as GUIs. Porém, para o contexto web, um modelo WUI pode ser definido como:

Definição 5 (Modelo WUI) *Uma WUI é modelada como um conjunto de objetos (widgets)* $O = \{o_1, o_2, ..., o_m\}$ *e um conjunto de propriedades* $P = \{p_1, p_2, ..., p_l\}$ *desses objetos. O estado de uma WUI é o conjunto P de todas as propriedades de todos os objetos de O que a WUI contém. Um conjunto válido de estados iniciais é associado a cada WUI. Um conjunto de estados* S_I *é chamado de estados iniciais válidos para uma WUI particular se a WUI pode estar em qualquer estado* $S_i \in S_I$ *quando ele é invocado primeiro.*

O estado de uma WUI não é estático, pois eventos executados sobre ela podem alterar seu estado e são chamados de estados alcançáveis. Com isso, a execução de um evento sobre esse modelo leva a outro estado. Eventos são definidos como:

Definição 6 (Eventos) *Os eventos* $E = \{e_1, e_2, ..., e_n\}$ *associados a uma WUI são funções de um estado para outro estado. Eventos ocorrem como parte de uma sequência de eventos. Uma sequência legal de eventos de uma WUI é* $e_1; e_2; e_3; ...; e_n$*, onde* $e_i + 1$ *pode ser executada imediatamente após* e_i*.*

A notação de função $S_j = e(S_i)$ é utilizado para indicar que S_j é o estado resultante da execução do evento e em um estado S_i. Os eventos podem ser combinados em sequências. Essas sequências de eventos pode ser executáveis se:

Definição 7 (Sequência de Eventos Executável) $e_1; e_2; e_3; ...; e_n$ *é uma sequência de eventos executável para um estado* S_0*, sse existir uma sequência de estados* $S_0; S_1; ...; S_n$ *tal que* $S_i = e_i(S_{i-1})$*, para para* $i = 1, ..., n$*.*

Um conjunto de teste T é formado por $T = \{t_1, t_2, ..., t_m\}$, sendo $m \geq 1$. Os dados de teste, neste contexto, são definidos:

Definição 8 (Dado de Teste) *Um dado de teste t é um par* $(S_0, e_1; e_2; ...; e_n)$*, que consiste do estado* $S_0 \in S_I$*, chamado de estado inicial. t é uma sequência legal de eventos* $e_1; e_2; ...; e_n$*.*

Se o estado inicial de um dado de teste não é alcançado e/ou a sequência de eventos é ilegal, então o dado de teste é não executável.

As Figuras 3.6(a) e 3.6(b) mostram o site da Faculdade de Computação (FACOM), da Universidade Federal de Mato Grosso do Sul (UFMS), em um estado S_0 e uma sequência de eventos executável correspondente ao S_0. Estendendo a notação de função acima, $S_j = (e_1, e_2, ..., e_n)(S_i)$, onde $e_1, e_2, ..., e_n$ é um sequência de eventos executável, denota que S_j é o estado que resulta da execução da sequência específica de eventos a partir do estado S_i.

(a) *Site Institucional da FACOM–UFMS.*

(b) *Sequência de Eventos Executáveis.*

Figura 3.6: *Site X Sequência de Eventos.*

Dentro desse contexto, surge o conceito de controlabilidade, que requer que a WUI seja levada a um estado válido antes de iniciar os eventos. Com cada WUI está associado um conjunto distinto de estados, chamado de estados iniciais válidos. Um estado inicial válido é definido por:

Definição 9 (Estado Inicial Válido) *Um conjunto de estados* S_I *é chamado de conjunto de estados iniciais válido para uma WUI particular, sse a WUI pode estar em qualquer estado* $S_i \in S_I$ *quando é invocada primeiramente.*

Com base em uma WUI, $S_i \in S_I$, ou seja, em um estado inicial válido, novos estados podem ser obtidos por meio da ocorrência de eventos em S_i. Esses estados são chamados de estados "alcançáveis", a partir de S_i. Formalmente, um estado alcançável é definido como se segue.

Definição 10 (Estado Alcançável) *O estado S_j é um estado alcançável, sse qualquer $S_j \in S_I$ e existe uma sequência de eventos executável $e_x, e_y, ..., e_z$ tal que $S_j = (e_1, e_2, ..., e_n)(S_i)$ para qualquer $S_i \in S_I$.*

As restrições temporais e de sincronização são uma parte importante do comportamento de uma WUI. Um exemplo comum de uma restrição temporal, em um evento WUI, é o tempo máximo permitido para esse evento ser executado. Outras restrições, como restrições de sincronização, podem exigir que um objeto deve ser baixado completamente antes do próximo evento ser executado. Tais restrições temporais podem ser definidas para cada evento no dado de teste por uma sequência de sincronização/temporal.

Definição 11 (Sincronização/Temporal) *Uma sequência de sincronização/temporal $T_1; T_2; T_3; ...; T_n$ está associada a cada dado de teste WUI, onde cada T_i é um conjunto de restrições de sincronização/temporal no evento e_i.*

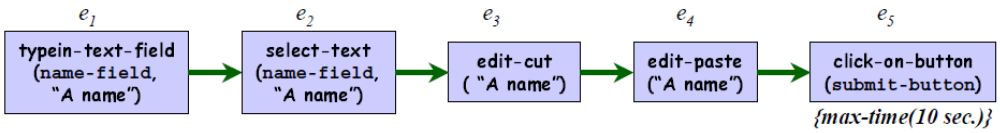


Figura 3.7: *Uma Sequência de Eventos WUI.*

Por exemplo, considere a sequência de eventos mostrada na Figura 3.7 para o WUI da Figura 3.3 (p. 55). O dado de teste consiste em 5 eventos, sendo os eventos e_1 e e_5 disponíveis na janela do navegador, enquanto que os eventos e_2, e_3 e e_4 são eventos disponibilizados pelo navegador. O evento e_5 tem uma restrição temporal que impõe um limite sobre o tempo decorrido entre a sua execução e a apresentação dos resultados. Se esse tempo for maior do que 10 segundos, em seguida, deve ser comunicado um erro. Esse controle torna muito mais complexo a determinação e execução de dados de teste.

3.5 Considerações Finais

Neste capítulo, foi discutida a abordagem de Testes Baseados em Modelos aplicada no contexto de teste WUI. O MBT é apresentado como uma solução para os problemas de automatização existentes na geração de casos de teste. Verificou-se que o MBT

pode ter um papel fundamental nos testes de software, garantindo uma maior qualidade do produto. Perante as características apresentadas, confirmou-se que é praticamente inviável gerar a mesma carga de testes que é possível obter com MBT de forma manual. As vantagens e limitações foram ressaltadas e algumas formalizações de conceitos WUI foram definidas. Este capítulo serve como fundamentação para entendimento dos conceitos pesquisados e apresentados no mapeamento sistemático descrito no Capítulo 4.

CAPÍTULO 4

Identificação do Estado da Arte em Testes GUI

Verificar a literatura pode ser complicado e trabalhoso devido à quantidade de influências que podem interferir, como linhas de pensamento, autores preferidos, grupos de pesquisa entre outros fatores. Para eliminar ou amenizar essas questões, é necessário seguir um método bem definido de busca e análise da literatura. O presente capítulo detalha o procedimento de mapeamento sistemático realizado no contexto de geração de dados de teste a partir da GUI que tem como foco identificar, catalogar e classificar os trabalhos existentes nesse contexto com o intuito de contribuir de forma substancial no seu entendimento e torná-los de fácil consulta e utilização pelo engenheiro de software.

4.1 Considerações Iniciais

A revisão sistemática da literatura (alguns momentos chamada de revisão sistemática) é o meio de identificar, interpretar e avaliar todas as pesquisas relevantes disponíveis para uma questão de pesquisa específica, área temática ou fenômeno de interesse (KITCHENHAM, 2004), considerando um período de tempo específico.

Diferentemente de revisões tradicionais, uma revisão sistemática é conduzida de acordo com um planejamento (ou protocolo) previamente definido (KITCHENHAM, 2004; BIOLCHINI et al., 2005). Esse planejamento é o ponto de partida para a revisão, cujos pontos principais são a definição de uma ou mais questões de pesquisa e dos métodos que serão empregados para conduzir a revisão, incluindo seleção de fontes para buscas e estratégias de busca. Além disso, uma revisão sistemática deve obrigatoriamente documentar esse protocolo de busca executado de forma a permitir que a revisão seja repetida por outros pesquisadores interessados.

Segundo Kitchenham et al. (2007), estudos individuais que contribuem para uma revisão sistemática são chamados de estudos primários; uma revisão sistemática é uma forma de estudo secundário. A condução de uma revisão sistemática supostamente apresenta uma avaliação justa do tópico de pesquisa à medida que utiliza uma metodologia de revisão rigorosa, confiável e passível de auditagem.

Em particular, pesquisadores, conduzindo revisões sistemáticas, devem despender esforços na identificação e relato de pesquisas que apoiam e não apoiam suas hipóteses. Se os estudos identificados apresentam resultados consistentes, a revisão sistemática provê indícios de que o fenômeno é robusto e generalizável a outros contextos. Caso os resultados dos estudos sejam inconsistentes, as fontes de variação desses resultados podem ser estudadas (MAFRA; TRAVASSOS, 2005).

O mapeamento sistemático (MS) é um tipo de revisão sistemática, no qual se realiza uma revisão mais ampla dos estudos primários, em busca de: identificar quais evidências estão disponíveis; identificar lacunas no conjunto dos estudos primários, direcionando o foco de revisões sistemáticas futuras; e identificar áreas nas quais mais estudos primários precisam ser conduzidos (KITCHENHAM, 2004). O estudo do MS fornece uma visão geral de uma área de pesquisa, identificando a quantidade, os tipos de pesquisas realizadas, os resultados disponíveis, além das frequências de publicações ao longo do tempo para identificar tendências (PETERSEN et al., 2008).

Há muitas razões para a realização de um MS, sendo os motivos mais comuns:

- Resumir as evidências existentes em relação a um tratamento ou tecnologia, por exemplo, resumir evidência empírica dos benefícios e limitações de um método específico;
- Identificar as eventuais lacunas existentes na pesquisa atual, a fim de sugerir futuras áreas de investigação; e
- Fornecer uma visão geral/subsídios que permitem avançar o conhecimento na investigação de novas áreas.

No entanto, mapeamentos sistemáticos também podem ser utilizados para examinar a extensão em que a evidência empírica suporta/contradiz hipóteses teóricas, ou até mesmo para auxiliar a geração de novas hipóteses (KITCHENHAM et al., 2007). De acordo com Kitchenham (2004), uma mapeamento sistemático é composto pelas etapas de planejamento, condução e relatório da revisão.

No planejamento, são descritas as necessidades que o pesquisador têm para a realização do mapeamento e como será o procedimento para sua condução. Partindo da motivação para a realização de um processo mais sistemático de varredura da literatura, questões de pesquisa são definidas e o protocolo da revisão é desenvolvido com a finalidade de responder a elas (ENGSTRöM; SKOGLUND; RUNESON, 2008). O protocolo congrega todos os passos a serem seguidos para a realização do mapeamento sistemático. Ele contempla as questões de pesquisa, as fontes de informação, critérios de consulta, critérios de seleção de fontes e também as ameaças a validade da pesquisa.

A etapa de condução se remete aos resultados da execução dos passos do protocolo. Nela são elencados quais trabalhos foram removidos em cada etapa da revisão

e os motivos para tal, os artigos reunidos e remanescentes ao final de cada etapa e os dados extraídos de cada obra. Tais dados podem ser o autor, veículo, ano de publicação, formas de avaliação da solução proposta ou mesmo a metodologia utilizada, podendo utiliza-los para, além de responder às perguntas de pesquisa, traçar perfis das pesquisas como quais os pesquisadores envolvidos, qual o recorte temporal no qual as pesquisas foram conduzidas e quais os veículos de publicação mais utilizados (KITCHENHAM, 2004). No relatório do mapeamento, etapa final, deve-se apresentar os resultados em um artigo para um periódico ou mesmo num documento de tese (KITCHENHAM, 2004).

4.2 Planejamento do Mapeamento Sistemático

De acordo com Kitchenham (2004), o planejamento de um mapeamento sistemático descreve o protocolo que foi estabelecido, sendo necessário especificar os seguintes elementos:

1. Questões de pesquisa;
2. Estratégia e execução de busca;
3. Critérios de inclusão e exclusão; e
4. Extração de dados e métodos de síntese.

O planejamento do mapeamento sistemático foi realizado a partir adaptação do modelo de protocolo apresentado por Petersen et al. (2008), cujos passos são ilustrados na Figura 4.1. Como pode ser observado na figura, os elementos apresentados por Kitchenham (2004), citados anteriormente, são relacionados ao protocolo, indicando quais fases englobam tais elementos. Essas fases foram conduzidas no período de 06 de fevereiro de 2012 a 01 de maio de 2013.

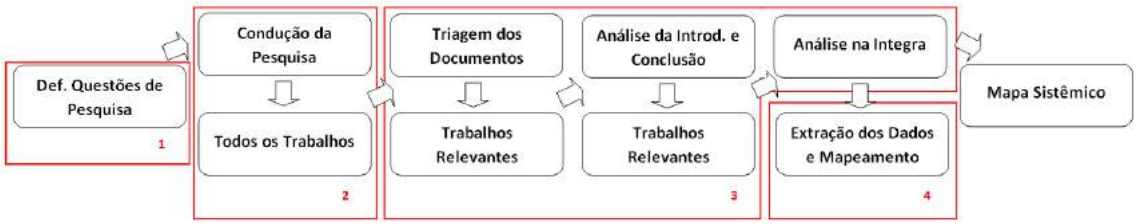


Figura 4.1: *Arcabouço para Mapeamento Sistemático (adaptado de Petersen et al. (2008)).*

4.2.1 Questões da Pesquisa

Com o objetivo de encontrar estudos primários para entender e sumarizar evidências sobre a adoção em conjunto de práticas de técnicas para geração automática de dados de teste a partir de GUI, as questões de pesquisa (QP) a seguir foram estabelecidas:

- **QP**$_1$: As técnicas de teste de software empregadas no teste GUI visam à cobertura de algum critério de teste específico?
  QP$_{1.1}$: Quais os critérios?
- **QP**$_2$: Quais meta-heurísticas, técnicas, algoritmos e estratégias são utilizadas para geração automática de dados de teste a partir de GUI?
- **QP**$_3$: As técnicas para geração automática de dados de teste GUI necessitam de um modelo de dados que abstraia a interface para realizar a geração?
  QP$_{3.1}$: O modelo é gerado automaticamente ou manualmente?
- **QP**$_4$: Quais ferramentas existem e como apoiam a geração automática de dados de teste utilizando-se GUI?
- **QP**$_5$: Em que domínios são aplicadas geração automática de dados de teste baseadas em GUI?

Cada questão será respondida e analisada sob diferentes pontos de vistas:

População: grupo populacional que será observado – trabalhos que discutem da utilização de técnicas para geração automática de dados de teste a partir de GUI;

Intervenção: o que será avaliado neste conjunto – o uso de técnicas para geração automática de dados de teste a partir de GUI;

Comparação: elementos que servirão como base de comparação – não aplicável;

Controle: elementos que servirão de base para avaliação das *strings* de busca – para avaliar as *strings* de busca foram identificados, por especialistas do domínio de interesse, 10 trabalhos considerados relevantes ao contexto dessa pesquisa, que são listados na Tabela 4.1. Esses trabalhos foram utilizados como artigos de controle que forneceram indícios de que a *string* de busca esta adequada, uma vez que todos esses artigos foram retornados após a aplicação das buscas; e

Resultados: Informações de saída esperadas com a pesquisa - conjunto de características bases para a utilização de técnicas para geração automática de dados de teste a partir de GUI, obtendo assim um conhecimento sobre o estado da arte da utilização destes dois conceitos juntos com o foco na geração automática de dados de teste, visando encontrar possíveis áreas que carecem de pesquisa a serem exploradas em trabalhos futuros.

4.2.2 Estratégia para a Execução da Busca

Revisões sistemáticas da literatura e estudos de mapeamento sistemático são formas relativamente novas de estudos secundários em Engenharia de Software (CHEN; SHEN, 2010). Identificar documentos relevantes de diversas fontes (bases) de dados eletrônicas é uma das atividades fundamentais da condução desses tipos de estudos.

Tabela 4.1: *Artigos de Controle.*

#	**Título**	**Referência**	**Base de Dados**
AC1	AutoBlackTest: Automatic Black-Box Testing of Interactive Applications	(MARIANI et al., 2012)	IEEE
AC2	Apply ant colony to event-flow model for graphical user interface test case generation	(HUANG; LU, 2012)	IEEE
AC3	Automated GUI Test Coverage Analysis Using GA	(RAUF et al., 2010a)	SCOPUS
AC4	Hierarchical GUI test case generation using automated planning	(MEMON; POLLACK; SOFFA, 2001a)	IEEE
AC5	Iterative execution-feedback model-directed GUI testing	(YUAN; MEMON, 2010b)	SCIENCE DIRECT
AC6	Using ontology to generate test cases for GUI testing	(LI et al., 2011)	ACM
AC7	Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback	(YUAN; MEMON, 2010a)	IEEE
AC8	Behind the Scenes: An Approach to Incorporate Context in GUI Test Case Generation	(ARLT; BERTOLINI; SCHäF, 2011)	IEEE
AC9	An approach to automatic input sequence generation for GUI testing using ant colony optimization.	(BAUERSFELD; WAPPLER; WEGENER, 2011)	SCOPUS
AC10	PETTool: A pattern-based GUI testing tool	(CUNHA et al., 2010)	IEEE

Segundo Chen e Shen (2010), essas fontes podem ser classificadas em duas categorias principais: motores de índice e sites dos editores. Os motores de índice trabalham com publicações de várias editoras. Pode-se citar como exemplo de motor de índice o *SCOPUS*. Os sites de editoras referem-se às bases de dados de literatura on-line fornecidas pelos editores para facilitar a recuperação da literatura publicada. Um site de editora popular na área da computação é o *IEEE*. Porém, como é o caso da *ACM*, algumas dessas fontes se enquadram nas duas categorias.

A estratégia de busca engloba a seleção das fontes de estudos primários e a construção da *string* de busca. A Tabela 4.2 apresenta as bases selecionadas como fontes de estudos primários para este mapeamento sistemático. As bases internacionais escolhidas são consideradas eficientes na condução de revisões sistemáticas no contexto de Engenharia de Software (DYBA; DINGSOYR; HANSSEN, 2007; ALI et al., 2010).

Tabela 4.2: *Bases de Consulta.*

Fonte de Busca	**Endereço**
ACM	http://portal.acm.org/dl.cfm
IEEE	http://ieeexplore.ieee.org/Xplore/dynhome.jsp
Science Direct / Elsevier	http://www.sciencedirect.com/
Scopus	http://www.scopus.com/

Para a construção da *string* de busca, foram selecionados conceitos chaves que se deseja investigar. A partir disso, os sinônimos, termos relacionados e siglas foram identificados:

- Interface Gráfica do Usuário

  Graphical user interface, GUI, *web application*, *web applications*; e
- Geração Automatizada de Dados de Teste

  Test data generation, *test-data generation*, *generating test data*, *generate test data*, *automated testing*, *automation testing* e *automation test*.

Baseando-se nos conceitos chaves acima, a *string* padrão de busca foi construída usando os conectores booleanos AND/OR:

("graphical user interface" OR "GUI" OR "web application" OR "web applications") AND ("test data generation" OR "test-data generation" OR "generating test data" OR "generate test data" OR "automated testing" OR "automation testing" OR "automation test")

4.2.3 Critérios de Inclusão e Exclusão

Os Critérios de Inclusão (CI) foram escolhidos de forma a permitir que estudos primários relevantes sejam incluídos na pesquisa; e os Critérios de Exclusão (CE), elencados para que seja possível descartar os estudos primários irrelevantes no contexto deste mapeamento.

Os critérios de inclusão são:

- **CI_1:** O estudo apresenta um estudo de caso ou relato de experiência de uso de técnicas para geração de dados de teste a partir de GUI;
- **CI_2:** O estudo apresenta uma investigação de características das técnicas para geração de dados de teste a partir de GUI;
- **CI_3:** O estudo propõe métodos de avaliação de técnicas para geração de dados de teste a partir de GUI; e
- **CI_4:** O estudo apresenta ferramentas que utilizam técnicas para geração de casos de teste a partir de GUI.

Os critérios de exclusão são:

- **CE_1:** O trabalho não está relacionado a nenhuma das questões de pesquisa;
- **CE_2:** O trabalho foi selecionado por outra string de busca aplicada a mesma base;
- **CE_3:** Falta de informação a respeito do trabalho;
- **CE_4:** O trabalho já foi selecionado por meio de outra fonte; e

- **CE_5:** O trabalho não está no idioma Português ou Inglês.

Baseando-se nos critérios de inclusão e exclusão, três etapas foram definidas para a seleção de trabalhos. A primeira foi baseada nas palavras-chaves, título e resumo dos trabalhos para concluir se o trabalho pode ou não ser incluído. Na segunda etapa a introdução e a conclusão foram considerado para análise e, na terceira, a análise foi aplicada sobre o trabalho como um todo.

4.2.4 Extração de Dados e Métodos de Síntese

De posse de todos os trabalhos selecionados, algumas informações poderão ser coletadas, como:

1. Qual a quantidade de trabalhos por autor;
2. Qual a quantidade de trabalhos por ano;
3. Quais as técnicas aplicadas para geração dos dados de teste;
4. Qual trabalho é mais citado pelos demais; e
5. Se existe alguma relação de auto-citação.

4.3 Condução do Mapeamento Sistemático

4.3.1 Definição das *Strings* de Busca

Com a finalidade de atender às particularidades de cada base de busca, a string padrão sofreu alterações, mas sem destoar com os protocolo de pesquisa construído.

Na máquina de busca da *ACM* foi utilizada a opção de busca avançada ("*Advanced Search*") e a string padrão foi reformulada com a separação lógica dos termos pelos operadores básicos OR e AND de acordo com a semântica da máquina. Foram executadas duas *string* diferentes, uma utilizando o comando *Abstract*, que busca apenas pelo texto do resumo (Figura 4.2) e outra com o comando *Title*, que foca a busca no título do documento (Figura 4.3).

```
((Abstract: "graphical user interface" ) OR (Abstract: "GUI") OR (Abstract: "web application")
OR (Abstract: "web applications")) AND ((Abstract: "test case generation") OR (Abstract:
"test-case generation") OR (Abstract: "generating Test Cases") OR (Abstract: "generate test
cases") OR (Abstract: "automated testing") OR (Abstract: "automation testing") OR (Abstract:
"automation test"))
```

Figura 4.2: *Primeira String Aplicada na Máquina de Busca da* ACM.

O *IEEEXplore* possui quatro tipos de busca *Basic*, *Advanced*, *Author* e *CrossRef*. Utilizou-se o módulo de busca *Advanced* e a opção "*Command Search*" foi possível

(Title: “graphical user interface”) OR (Title: “GUI”) OR (Title: “web application”) OR (Title: “web applications”)) AND ((Title: “test case generation”) OR (Title: “test-case generation”) OR (Title: “generating Test Cases”) OR (Title: “generate test cases”) OR (Title: “automated testing”) OR (Title: “automation testing”) OR (Title: “automation test”))

Figura 4.3: *Segunda String Aplicada na Máquina de Busca da* ACM.

inserir o operador AND e OR entre as palavras-chave. A *string* padrão foi reformulada utilizando o comando *Abstract*, que busca apenas pelo texto do resumo (Figura 4.4) e outra com o comando *Document Title*, que foca a busca no título do trabalho (Figura 4.5).

((“Abstract”: “graphical user interface”) OR (“Abstract”: “GUI") OR (“Abstract”: “web application”) OR (“Abstract”: “web applications”)) AND ((“Abstract”: “test case generation”) OR (“Abstract”: “test-case generation”) OR (“Abstract”: “generating Test Cases”) OR (“Abstract”: “generate test cases”) OR (“Abstract”: “automated testing”) OR (“Abstract”: “automation testing”) OR (“Abstract”: “automation test”))

Figura 4.4: *Primeira String Aplicada na Máquina de Busca da* IEEE.

((“Document Title”: “graphical user interface”) OR (“Document Title”: “GUI”) OR (“Document Title”: “web application”) OR (“Document Title”: “web applications”)) AND ((“Document Title”: “test case generation”) OR (“Document Title”: “test-case generation”) OR (“Document Title”: “generating Test Cases”) OR (“Document Title”: “generate test cases”) OR (“Document Title”: “automated testing”) OR (“Document Title”: “automation testing”) OR (“Document Title”: “automation test”))

Figura 4.5: *Segunda String Aplicada na Máquina de Busca da* IEEE.

Utilizando a máquina de busca da *SCIENCE* foi selecionada a opção de busca avançada (*Advanced Search*”), restringindo a busca apenas a periódicos sobre o assunto Computação *Computer Science*. A *string* utilizada é representada na Figura 4.6.

({graphical user interface} OR {gui} OR {web application} OR {web applications}) AND ({test case generation} OR {test-case generation} OR {generating Test Cases} OR {generate test cases} OR {automated testing} OR {automation testing} OR {automation test})

Figura 4.6: *String Aplicada na Máquina de Busca da* SCIENCE.

Na máquina de busca da *SCOPUS* foram utilizados os comandos *TITLE-ABS-KEY*, que restringe a busca no título, resumo e palavras-chaves, além do comando *SUBAREA*, que restringiu a busca do contexto de computação. A *string* utilizada é representada na Figura 4.7.

Na próxima seção, são apresentados os números de trabalhos localizados com a aplicação das *strings* de busca.

TITLE-ABS-KEY(({graphical user interface} OR {gui} OR {web application} OR {web applications}) AND ({test case generation} OR {test-case generation} OR {generating Test Cases} OR {generate test cases} OR {automated testing} OR {automation testing} OR {automation test})) AND SUBJAREA(comp)

Figura 4.7: *String Aplicada na Máquina de Busca da SCOPUS.*

4.3.2 Resultados da Busca

Na Tabela 4.3 é apresentado o número de artigos retornados para cada *string* de busca em cada fonte selecionada.

4.3.3 Seleção Preliminar dos Trabalhos

A seleção preliminar dos trabalhos teve o suporte computacional da ferramenta *JabRef* (BAOLA; NEURONADE, 2016), que se trata de um gerenciador de referências baseado em bases de dados *BibTex* e que auxiliou nas atividades de organização e catalogação dos documentos.

Para a seleção dos trabalhos, optou-se por aplicar três etapas:

1. Leitura do título, das palavras-chave e do resumo (*abstract*);
2. Leitura da introdução e conclusão; e
3. Leitura completa dos trabalhos.

Na primeira etapa, utilizando-se a ferramenta *JabRef*, cada um dos trabalhos foi analisado por dois especialistas aplicando-se todos critérios de inclusão e apenas os critérios de exclusão CE_1, CE_2 e CE_3 definidos na Seção 4.2.3. Essa análise ocorreu na ordem de aplicação das *strings* de busca e foram obtidos os dados apresentados na Tabela 4.3.

Tabela 4.3: *Resultado da Primeira Análise dos Trabalhos.*

Strings	CI1	CI2	CI3	CI4	CE1	CE2	CE3	Total CI (%)	Total CE (%)	TOTAL
ACM-1	21	3	1	2	87	0	1	27 (23,48%)	88 (76,52%)	115
ACM-2	2	0	0	0	7	15	0	2 (8,34%)	22 (91,66%)	24
IEEE-1	20	3	0	2	53	0	2	25 (31,25%)	55 (68,75%)	80
IEEE-2	2	0	0	0	5	19	0	2 (7,69%)	24 (92,30%)	26
SCIENCE	5	1	0	0	151	0	25	6 (3,30%)	176 (96,70%)	182
SCOPUS	31	2	1	2	134	0	1	36 (21,05%)	135 (78,95%)	171
TOTAL	81	9	2	6	437	34	29	98 (16,39%)	500 (83,61%)	598

Observa-se um grande número de trabalhos selecionados pelos critérios de inclusão, 98 trabalhos que representam 16,39% do total. Porém, os trabalhos redundantes entre as bases de busca ainda não haviam sido identificados, ou seja, o critério CI_4 ainda não havia sido aplicado. Após essa identificação e a leitura da introdução e da conclusão (segunda etapa) esse número de trabalhos selecionados reduziu para 59, que correspondem a 9,87% do total como mostra a Tabela 4.4

Tabela 4.4: *Resultado da Segunda Análise dos Trabalhos.*

Strings	CI1	CI2	CI3	CI4	CE1	CE2	CE3	CE4	Total CI (%)	Total CE (%)	TOTAL
ACM-1	21	3	1	2	87	0	1	0	27 (23,48%)	88 (76,52%)	115
ACM-2	2	0	0	0	7	15	0	0	2 (8,34%)	22 (91,66%)	24
IEEE-1	10	1	0	1	53	0	2	13	12 (15,00%)	68 (85,00%)	80
IEEE-2	1	0	0	0	5	19	0	1	1 (3,84%)	25 (96,16%)	26
SCIENCE	4	1	0	0	151	0	25	1	5 (2,75%)	177 (97,25%)	182
SCOPUS	11	0	1	0	134	0	1	24	12 (7,02%)	159 (92,13%)	171
TOTAL	49	5	2	3	437	34	29	39	59 (9,87%)	539 (90,13%)	598

No processo de seleção final, terceira etapa, os estudos foram analisados completamente e, posteriormente, 39 estudos primários foram selecionados para compor o mapeamento, 6,52% dos 598 estudos primários inicialmente selecionados. Esses trabalhos são listados na Figura 4.5 e observa-se que esta taxa de redução é coerente com outras pesquisas na área (PETERSEN et al., 2008; ENGSTRÃM; RUNESON, 2011).

4.3.4 Análise Final dos Trabalhos Selecionados

Em relação à primeira questão de pesquisa QP_1, as trabalhos não identificam um critério de teste específico, mas às vezes mencionam que a técnica é usada para geração de dados de teste. Entre as três técnicas de testes mais conhecidas, a funcional, estrutural e baseadas em defeitos, a primeira corresponde a 94,8% dos trabalhos, uma vez que quando o estudo não mencionou qual técnica foi usada foi considerado o uso da técnica funcional, por se tratar de geração de dados de teste baseando-se em GUI. Algumas obras, no entanto, além da técnica funcional também usam o código-fonte (técnica estrutural) para orientar a sua técnica ou metodologia de pesquisa para a geração de dados de teste (XIE; MEMON, 2008; LI et al., 2009; ARLT; BERTOLINI; SCHäF, 2011; PENG; LU, 2011).

Para responder à questão de pesquisa QP_2, foram analisados os meta-heurísticas e técnicas geralmente utilizadas em estudos para a geração de dados de teste. Nesse caso, a distribuição das obras foi mais uniforme, com uma ligeira maioria de 10 estudos (25,6%) utilizando algoritmos genéticos (ALSMADI; MAGEL, 2007; KUK; KIM, 2008; YUAN; COHEN; MEMON, 2009; ALSMADI, 2010; HUANG; COHEN; MEMON, 2010; RAUF et al., 2010a; PENG; LU, 2011; RAUF; JAFFAR; SHAHID, 2011). Outra meta-heurística utilizada em outros três estudos foi a Colônia de Formigas (LU et al., 2008; BAUERSFELD; WAPPLER; WEGENER, 2011; HUANG; LU, 2012).

Também foram utilizados outras meta-heurísticas para a geração de dados de teste, como nos estudos de caso de Rauf et al. (2010a) e Becce et al. (2012), que se aplicavam *Q-Lerning*, uma ferramenta da área da IA que aprende a interagir com o aplicativo em teste e estimular a sua funcionalidade, e do trabalho de Rauf et al. (2010b), que usou Enxame de Partículas. Além de meta-heurísticas, outras técnicas são identificadas como no trabalhos de Li et al. (2011) e Li et al. (2009), os quais usaram ontologias.

Tabela 4.5: *Relação dos Artigos Selecionados.*

id	Título do Trabalho	Referência
1	A Framework for GUI Testing Based on Use Case Design	(BERTOLINI; MOTA, 2010)
2	An approach to automatic input sequence generation for GUI testing using ant colony optimization	(BAUERSFELD; WAPPLER; WEGENER, 2011)
3	An event interaction structure for GUI test case generation	(QIAN; JIANG, 2009)
4	A new approach for session-based test case generation by GA	(PENG; LU, 2011)
5	An Ontology-Based Approach for GUI Testing	(LI et al., 2009)
6	Apply ant colony to event-flow model for graphical user interface test case generation	(HUANG; LU, 2012)
7	A test automation solution on gui functional test	(XIAOCHUN et al., 2008)
8	AutoBlackTest: Automatic Black-Box Testing of Interactive Applications	(MARIANI et al., 2012)
9	Automated GUI test coverage analysis using GA	(RAUF et al., 2010a)
10	Automated GUI Testing for J2ME Software Based on FSM	(HOU; CHEN; DU, 2009)
11	Automated gui testing guided by usage profiles	(BROOKS; MEMON, 2007)
12	Automated optimum test case generation using web navigation graphs	(SHAHZAD et al., 2009)
13	Automatic Generation of Testing Environments for Web Applications	(KUK; KIM, 2008)
14	Behind the Scenes: An Approach to Incorporate Context in GUI Test Case Generation	(ARLT; BERTOLINI; SCHäF, 2011)
15	Covering array sampling of input event sequences for automated gui testing	(YUAN; COHEN; MEMON, 2007)
16	Development of an Improved GUI Automation Test System Based on Event-Flow Graph	(LU et al., 2008)
17	Extracting widget descriptions from GUIs	(BECCE et al., 2012)
18	Fully automated gui testing and coverage analysis using genetic algorithms	(RAUF; JAFFAR; SHAHID, 2011)
19	Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback	(YUAN; MEMON, 2010a)
20	Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences	(WHITE; ALMEZEN, 2000)
21	GUI Interaction Testing: Incorporating Event Context	(YUAN; COHEN; MEMON, 2011)
22	GUI path oriented test generation algorithms	(ALSMADI; MAGEL, 2007)
23	GUI test-case generation with macro-event contracts	(CHEN; SHEN, 2010)
24	GUI Testing Techniques Evaluation by Designed Experiments	(BERTOLINI et al., 2010)
25	Hierarchical GUI Test Case Generation Using Automated Planning	(MEMON; POLLACK; SOFFA, 2001b)
26	IDATG: an open tool for automated testing of interactive software	(BEER; MOHACSI; STARY, 1998)
27	Intra Component GUI Test Case Generation Technique	(HAYAT; QADEER, 2007)
28	Iterative execution-feedback model-directed GUI testing	(YUAN; MEMON, 2010c)
29	PETTool: A pattern-based GUI testing tool	(CUNHA et al., 2010)
30	PSO based test coverage analysis for event driven software	(RAUF et al., 2010b)
31	Repairing GUI Test Suites Using a Genetic Algorithm	(HUANG; COHEN; MEMON, 2010)
32	Test case generator for GUITAR	(HACKNER; MEMON, 2008)
33	Towards Dynamic Adaptive Automated Test Generation for Graphical User Interfaces	(YUAN; COHEN; MEMON, 2009)
34	User behavior augmented software testing for user-centered GUI	(CHUANG; SHIH; HUNG, 2011)
35	Using a goal-driven approach to generate test cases for GUIs	(MEMON; POLLACK; SOFFA, 1999)
36	Using a pilot study to derive a GUI model for automated testing	(XIE; MEMON, 2008)
37	Using Genetic Algorithms for test case generation and selection optimization	(ALSMADI, 2010)
38	Using GUI Run-Time State as Feedback to Generate Test Cases	(YUAN; MEMON, 2007)
39	Using ontology to generate test cases for GUI testing	(LI et al., 2011)

Um fato que foi observado, e que deve ser explorado, é que alguns estudos (HOU; CHEN; DU, 2009; RAUF et al., 2010a; ARLT; BERTOLINI; SCHäF, 2011) necessitam de um modelo inicial da aplicação referente a sua GUI para executar e gerar dados de teste (questão de pesquisa QP_3). Apenas o trabalho de Mariani et al. (2012) empregou a geração automática de um modelo e produziu os dados de teste de forma incremental, percorrendo o modelo GUI do aplicativo em teste e sem necessidade de intervenção humana.

Respondendo à quarta questão de pesquisa QP_4, algumas ferramentas que ajudam na geração de dados de teste foram identificadas durante o mapeamento. A maioria das ferramentas são complementares, isto é, elas permitem a obtenção de melhores resultados quando combinadas. Uma das ferramentas mais utilizada nos estudos selecionados

foi *GUITAR* (HACKNER; MEMON, 2008), que foi utilizada em 42% dos estudos selecionados e terá seu funcionamento detalhado na Seção 4.4.2.

Finalmente, respondendo à questão de pesquisa QP_5, a maioria dos estudos selecionados, aproximadamente 95%, são aplicados em sistemas *desktop*. Somente os estudos Peng e Lu (2011), Shahzad et al. (2009) e Kuk e Kim (2008) aplicam-se a proposta no contexto web, mostrando assim que muito ainda pode ser feito nesta área.

A Figura 4.8 utiliza um grafo dirigido para apresentar a ligação cronológica existente entre os 39 estudos primários selecionados. As setas indicam que um dado trabalho referencia outro. Observe que, de 1998 até a data de aplicação da pesquisa, a maioria dos estudos se concentra no ano de 2010, totalizando 10 estudos. No entanto, o único estudo preliminar identificado em 2001 foi citado por 15 outras obras, e todos os estudos a partir de 2010 são referenciados juntos por outros 8 estudos. Dois estudos que podem ser considerados referências são a um de autoria de White e Almezen (2000) e um de autoria de Memon, Pollack e Soffa (1999) com 11 e 15 citações cada, respectivamente. Os 39 estudos primários selecionados envolveram 74 autores diferentes de 27 afiliações (instituições) distribuídas em 13 países. A maioria dos trabalhos neste contexto são de autoria dos mesmos autores. Um exemplo é o Dr. Atif Memon com a participação em, aproximadamente, 31% dos estudos selecionados e pode ser considerado uma referência na geração de dados de teste GUI. Destaque para a *University of Maryland* nos EUA, aparecendo como uma instituição e país com mais participação nos estudos selecionados, 12 (30,8%) e 15 (38,5%), respectivamente.

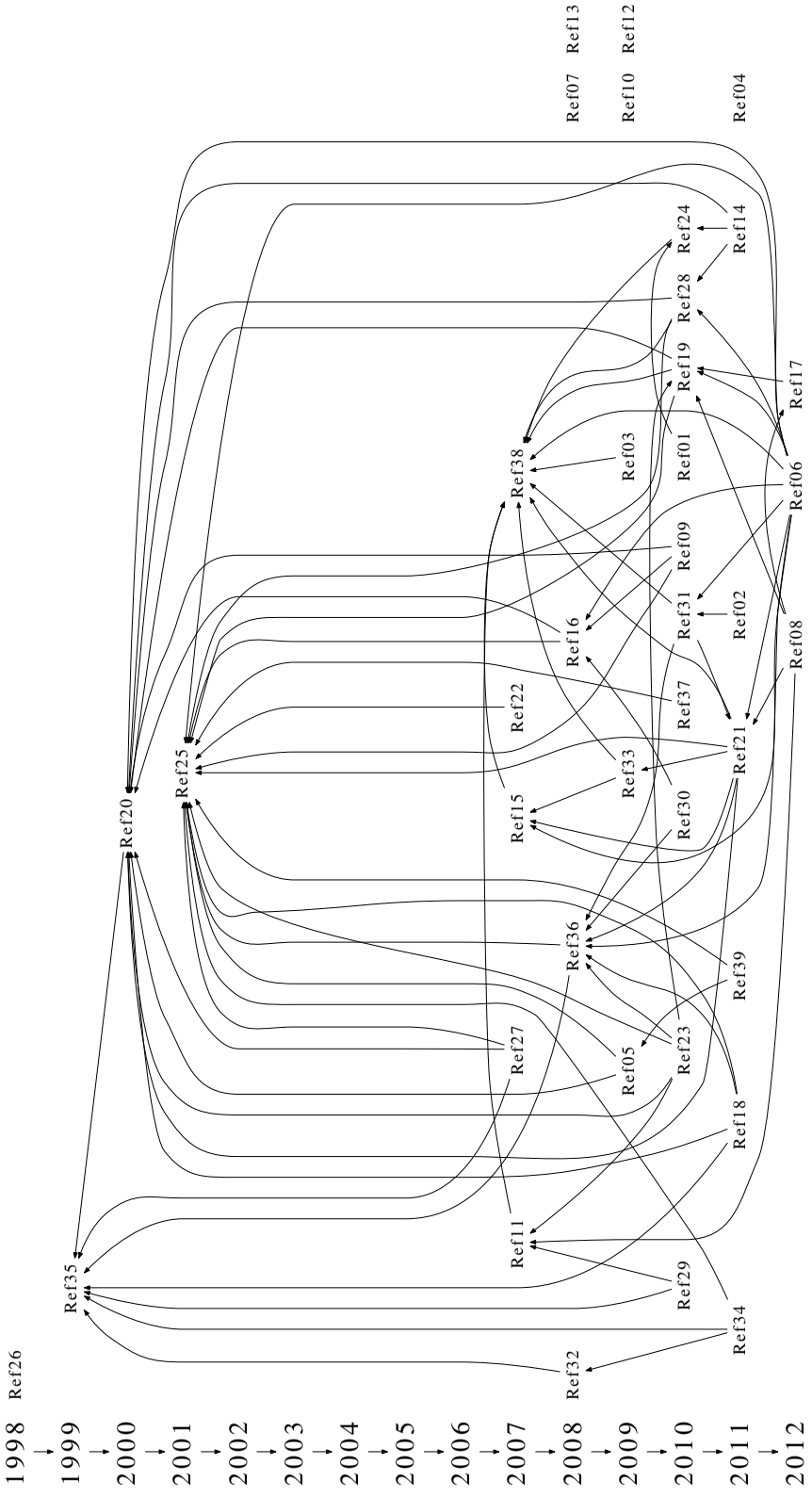


Figura 4.8: *Distribuição e Citação Entre os Estudos Primários.*

4.4 Descrição de Alguns Trabalhos

É difícil deduzir o comportamento de uma aplicação gráfica a partir de seu código-fonte sem executá-lo, isso porque os *widgets* são muitas vezes acessíveis apenas a partir de um determinado estado ou com a satisfação de restrições. A relação entre controles da GUI e seus respectivos eventos, e até mesmo a estrutura da GUI, podem ser definidos dinamicamente em tempo de execução, com apenas um esqueleto básico da GUI definido estaticamente no código-fonte (SILVA; CAMPOS, 2013). A análise dinâmica que envolve a execução da aplicação e o comportamento de execução da interface gráfica é mais adequada para a extração de modelos, mas mais difícil de automatizar (GRILO; PAIVA; FARIA, 2010). Para se fazer a execução automática de uma aplicação gráfica é necessário simular um usuário final por meio da sua interface gráfica. A análise dinâmica permite modelar o comportamento de alterações nas GUIs, por exemplo, quando a visibilidade de um objeto depende do estado de outro objeto (AHO; RATY; MENZ, 2013).

Um grande desafio em percorrer a GUI durante a análise dinâmica automaticamente está em fornecer entradas para a aplicação específicas para os campos de entrada sem instruções pré-definidas do usuário (AHO; MENZ; RATY, 2011). Geralmente, alguma intervenção humana, tais como o fornecimento de um nome de usuário e senha válidos para uma tela de login, é necessária durante o processo de modelagem para atingir todas as partes do GUI e conseguir uma cobertura razoável com modelos extraídos de forma dinâmica (AHO; MENZ; RATY, 2011). Outra opção é que um especialista aplique avaliações manualmente, corrija ou estenda os modelos extraídos. A eficiência destas técnicas de modelagem semi-automáticas depende em grande parte do grau de intervenção humana (KULL, 2012).

Um dos trabalhos mais importantes identificado pelo MS, apresentado na Seção 4, é o trabalho de Mariani et al. (2012). A técnica desenvolvida e a ferramenta utilizada são denominadas *AutoBlackTest*, que trabalha com a geração de um modelo e produz os dados de teste de forma incremental ao interagir com o aplicativo em teste. Para isso, é utilizado o *Q-Lerning*, da área de Inteligência Artificial, que aprende a interagir com o aplicativo em teste e estimular as suas funcionalidades. Uma característica importante desse trabalho é que o *AutoBlackTest* não depende de um conjunto inicial para execução. A grande maioria das técnicas atuais depende desse conjunto e, para gerar dados de teste GUI, trabalha em duas fases (YUAN; COHEN; MEMON, 2011; MEMON; POLLACK; SOFFA, 2001b): gera um modelo das sequências de eventos que podem ser produzidos por meio da interação com a interface gráfica do aplicativo em teste e gera um conjunto de dados de teste que cobre as sequências no modelo.

A eficácia dessas técnicas depende da integridade do modelo inicial. Quando o modelo inicial é obtido pela estimulação do aplicativo em teste com uma estratégia de

amostragem simples que utiliza um subconjunto de ações GUI para navegar e analisar as janelas, o modelo derivado é parcial e incompleto (MARIANI et al., 2012). Com isso, os dados de teste gerados podem ignorar muitas interações e janelas não descobertas na fase inicial. Para avaliar a proposta, Mariani et al. (2012) conduziram uma avaliação empírica comparativa entre o *AutoBlackTest* com a ferramenta *GUITAR*, utilizando quatro aplicações para computadores *desktop*. Foram aplicadas sessões de 12 horas de testes, mostrando que o *AutoBlackTest* pode gerar dados de teste que atingem uma maior cobertura do código e ainda revelar um maior número de falhas se comparado à *GUITAR*.

Outro trabalho que se desta é a pesquisa apresentada por Mesbah, Deursen e Lenselink (2012), que propõe uma abordagem para analisar automaticamente as mudanças de estado na interface de aplicações web com tecnologia *Ajax*. A abordagem é baseada em um *crawler* que exercita o código do lado do cliente e identifica os elementos clicáveis que alteram o estado dentro do *DOM* (do inglês, *Dynamic Document Object*) construído dinamicamente no navegador. A partir dessas mudanças de estado, é inferido um grafo de fluxo de estado que captura os estados GUI e as possíveis transições entre eles. Essa abordagem possui suporte de uma ferramenta de código aberto, denominada *CRAWLJAX*.

Aho et al. (2013) apresentam uma abordagem e uma ferramenta, denominada *Murphy*, que permite extrair automaticamente modelos que podem ser usados no teste GUI. A ferramenta analisa a GUI ao interagir com a aplicação simulando um usuário final e gera uma Máquina de Estados Finitos baseada no modelo comportamental, observado durante a execução da aplicação. Aho et al. (2014) apresentam a avaliação da ferramenta *Murphy* em ambiente industrial que demonstrou que os modelos extraídos podem ser utilizados para automatizar e apoiar diversas atividades de teste, incluindo suporte a testes manuais GUI, proporcionando redução no tempo de execução dos casos de teste existentes. Também dentro desse mesmo contexto, Miao e Yang (2010b) propuseram uma representação por máquina de estado finito baseando-se no modelo de automação de teste GUI (*GuiTam*) e ferramentas para a construção automática dos modelos de estado por meio de análise dinâmica.

Em relação às técnicas que abordam testes baseado em modelos, destacam-se:

- Tradicional de TBM – que exploram alguma estratégia para geração de um modelo que é capaz de representar a aplicação web como um todo, incluindo as regras de navegação, caminhos possíveis e eventos. Geralmente, essas abordagens são dependentes do código ou ferramenta e é preciso uma quantidade considerável de tempo para se gerar modelos adequados; e
- PBGT (*Pattern-Based Graphical User Interface*) – utilizam uma linguagem específica de domínio (DSL – *Domain-Specific Language*). O PBGT tem como objetivo diminuir o esforço na construção de modelos para representarem as GUIs. Com ele é permitida a reutilização de padrões de interface do usuário por meio de estraté-

gias de teste (por exemplo, autenticação, entradas de dados e controles de login) que são convertidos em modelos. PBGT representa uma nova abordagem de testes GUI baseada em modelo e melhora as atuais abordagens de teste GUI (MOREIRA; PAIVA; MEMON, 2013; MOREIRA; PAIVA, 2014b).

4.4.1 *Gaps* para Automação de Testes GUI

No trabalho de Aho et al. (2015), são identificados *gaps* entre os métodos, ferramentas acadêmicas e requisitos industriais que estão dificultando a adoção da extração de modelos GUI para testes automatizados. Esses *gaps* (lacunas) estão relacionadas tanto com a extração automática dos modelos GUI como na utilização. São eles:

- G_1 – Ampliação para sistemas não-triviais, mantendo a precisão nos modelos extraídos;
- G_2 – Alcançar uma cobertura suficiente em um tempo razoável para o modelo extraído;
- G_3 – Validação da exatidão e cobertura dos modelos extraídos;
- G_4 – Aplicabilidade geral das ferramentas fornecidas;
- G_5 – O esforço da introdução e adoção;
- G_6 – Minimizando o esforço manual em teste de GUI; e
- G_7 – Minimizar o esforço de manutenção.

Os *gaps* G_1 até G_4 estão relacionados com extração automática de modelos de GUI e os *gaps* $G_5 – G_7$ estão relacionadas ao uso dos modelos extraídos para automatizar e apoiar o teste GUI. Como esse trabalho está inserido dentro do contexto do primeiro grupo, os *gaps* G_1 até G_4 serão detalhados.

Referente ao G_1, apesar da evolução de hardware e algoritmos utilizados na extração de modelo, a explosão no número de estados permanece como um desafio na criação de modelos. O desafio é encontrar o equilíbrio entre extrair modelos mais precisos e manter a complexidade computacional em um nível viável para que o modelo possa se utilizado em um tempo satisfatório. Compreende-se como estados GUI um conjunto de objetos e seus valores de propriedades e qualquer diferença no número de objetos ou valores de propriedade, podendo significar um estado diferente. Alguns valores de propriedade podem ter número enorme ou mesmo infinito de valores possíveis, que por sua vez faz com que o número de estados GUI seja enorme ou até infinito. Sem um método adequado para limitar a explosão desses estados se torna inviável a utilização de testes GUI baseados em modelos Aho et al. (2015). O desafio é encontrar o equilíbrio entre o aumento da expressividade para extrair modelos mais precisos e manter os modelos pequenos o suficiente para ser computacionalmente viável para o modelo de inferência e verificação do modelo (MEINKE; WALKINSHAW, 2012).

A solução para a redução dos estados do modelo é, claro, abstrair ou ignorar algumas das propriedades ou valores da GUI quando distinguir os estados do modelo, mas o desafio é como encontrar o nível certo de abstração e automaticamente escolher o propriedades e valores importantes. Uma solução eficiente para reduzir o número de estados GUI é ignorar os valores de dados, como texto em campos de entrada, e concentrar-se nas interações que estão disponíveis para o usuário final em cada estado GUI. Para capturar o contexto das interações realizadas, os valores de dados podem ser salvos nas propriedades das transições de estado, como em (AHO; MENZ; RATY, 2011). A desvantagem é que a redução nos estados resultará o aumento da quantidade de possíveis transições.

O grande desafio referente ao *gap* G_2 é em relação a exploração durante a extração automatizada do modelo GUI. O desafio é como fazer para ter acesso a todas as partes da GUI para se ter uma boa cobertura nos modelos extraídos. Por exemplo, é muito improvável se encontrar usuário e senha válidos para uma tela de *login* com algoritmos de geração aleatória, se o usuário não forneceu qualquer conjunto predefinido de dados de teste. Isso porque uma vez que existe a validade individual, também existe a dependência entre ambos. A geração aleatória de entrada pode ser usada para melhorar a cobertura de modelos extraídos, mas encontrar valores específicos com métodos aleatórios requer muito tempo, retardando o processo de extração modelo. Ao usar modelos extraídos para testes, as partes da GUI que estão faltando nos modelos não serão cobertos com os dados de teste automaticamente derivados dos modelos. Normalmente, o utilizador tem de fornecer combinações válidas de entrada antes ou durante o processo de extração modelo. A quantidade de esforço manual deve ser minimizada pelo fornecimento de apoio ferramental para o usuário. Outra opção seria usar análise estática do código-fonte para gerar dados significativos, mas para dados de autenticação, por exemplo, isso não se aplica, uma vez que nomes de usuários e senhas são normalmente armazenados em bancos de dados e não no código-fonte.

Se o utilizador tem que fornecer as combinações de entrada válidas durante o processo de extração do modelo, uma maneira prática é usar a GUI real da aplicação, como em (AHO; MENZ; RATY, 2011). No entanto, pode ser mais fácil fornecer a entrada para vários estados GUI, utilizando uma apresentação visual do modelo obtido, de modo que o usuário possa selecionar o estado e, em seguida, os *widgets* para a entrada. Com a ferramenta *Murphy* (AHO et al., 2014) todo o aplicativo de dados e instruções para extração modelo específico são armazenadas em um *script* que é usado para iniciar o processo automatizado de extração modelo. Na prática, um processo iterativo é usado para definir e melhorar os *scripts* para extrair modelos com cobertura suficiente.

Um objetivo comum, nesse contexto, é o de maximizar a cobertura de código-fonte. Por exemplo, o número de estados GUI cobertos enquanto se minimiza o tempo de

extração. Uma solução proposta é a utilização de várias estratégias de extração com base na classificação de *widgets* GUI, como em (AHO; RATY; MENZ, 2013), para selecionar as interações com maior probabilidade de resultar novos estados GUI.

Outro fator a considerar é o esforço manual exigido para alcançar a extração automatizada de alguns estados para o modelo. Alguns dos fluxos da GUI são difíceis de explorar, por exemplo, caixas de diálogo mostradas somente quando a conexão de rede é perdida. Portanto, cabe aos engenheiros de teste decidirem quando uma cobertura é suficiente. Por exemplo, caso 80% de todos os possíveis fluxos de uma GUI foram atingidos.

O *gap* G_3 discute a validação da exatidão e cobertura dos modelos extraídos. Com abordagens dinâmicas de engenharia reversa, os modelos extraídos são baseadas no comportamento observado do sistema implementado, ao invés do comportamento esperado definido em requisitos ou outras especificações. Isso leva a um desafio de utiliza-los para gerar automaticamente oráculos de teste significativos sem elaboração manual (AHO; MENZ; RATY, 2011). Algumas abordagens usam modelos extraídos para automatizar várias atividades de testes, mas normalmente os modelos gerados têm de ser inspecionados manualmente para que sejam validadas a correção ou adição de novos requisitos. Geralmente, o objetivo da automação dos teste é reduzir e evitar passos manuais, mas as abordagens de extração desses modelos devem fornecer meios práticos para validar e corrigir manualmente os modelos, de tal forma que, caso o modelo seja gerado novamente, as alterações manuais sejam preservadas.

A solução mais prática para validar os modelos extraídos GUI, segundo a literatura, parece ser a inspeção visual (AHO et al., 2014). Os modelos extraídos são ilustrados em um alto nível de abstração e os estados e as transições do modelo são registrados por *screenshots* da GUI real, de modo que a correção do modelo e o comportamento da aplicação GUI modelada possam ser inspecionados visualmente e validados pelo usuário com base em requisitos, projetos, ou outras especificações. Se o modelo extraído não representa todo o comportamento da GUI, tem que ser melhorado (relacionado com a G_2), possibilitando fornecer à ferramenta de extração as possíveis alterações/inclusões necessárias.

Para o *gap* G_4 a aplicabilidade geral das ferramentas utilizadas atualmente deve ser analisada. O problema com a maioria das abordagens de extração de modelo para aplicações GUI é que elas são limitadas a linguagens de programação ou plataformas específicas. É um desafio fornecer técnicas de engenharia reversa independentes da plataforma GUI e independente da linguagem de programação. A instrumentação fornecida pelos arcabouços GUI geralmente permite uma análise mais detalhada, sendo seu uso recomendado para fornecer suporte para as plataformas GUI mais comuns. Para a análise GUI, independente de plataforma, deve-se capturar e comparar imagens da tela antes

e depois de cada interação, procurando identificar as mudanças, que segundo Aho et al. (2013) parece ser a abordagem mais eficiente. Comparação das imagens de antes e depois da aplicação GUI pode ser usada para encontrar a janela GUI que deve ser analisada. Elementos GUI que têm interação podem ser automaticamente detectados e localizados, por exemplo, pela automatização do uso da tecla "tab" para percorrer os elementos habilitados e comparando os *screenshots* para encontrar áreas alteradas, tal como uma área delimitada ,selecionando o elemento em foco na tela. A estrutura e o comportamento da GUI podem ser analisados a partir das imagens dessa tela com base nas pistas que a aplicação GUI ou a plataforma (sistema operacional) oferece para o utilizador final, tal como a forma do cursor do mouse.

Perante os *gaps* apresentados, algumas pesquisas e ferramentas vêm sendo desenvolvidas. Um dos grupos de pesquisa que se destaca é o coordenado pelo professor Dr. Atif M. Memon[1]. O grupo trabalha no desenvolvimento e aplicação de ferramentas para teste GUI em alguns contextos. A ferramenta mais conhecida é a *GUITAR*[2] (NGUYEN et al., 2014) que trabalha com aplicações *Desktop*. Hoje o grupo tem focado suas pesquisas na ferramenta que trabalha com aplicativos móveis (AMALFITANO et al., 2014; AMALFITANO et al., 2015), tanto para Android (Android GUITAR[3] (AMALFITANO et al., 2012b)) como para IPhone (*IPhone GUITAR*[4]). Além dessas ferramentas, o grupo desenvolveu a ferramenta *Web GUITAR*[5], com foco em aplicações web. Porém, segundo contato feito via e-mail com o professor Memon, ele afirmou que as pesquisas utilizando essa ferramenta estão paradas uma vez que o grupo está trabalhando com as ferramentas do contexto *mobile* e até mesmo por algumas limitações da ferramenta que ainda não foram sanadas. A única pesquisa encontrada foi a de Meireles (2015), que trabalhou na melhoria de algumas limitações da ferramenta. Na Seção 4.4.2 as características e limitações dessa ferramenta são detalhadas.

4.4.2 *Web GUITAR – GUI Testing FrAmewoRk*

A *Web GUITAR*, referenciada neste texto pela sigla *WG*, é uma ferramenta que possibilita a geração e execução dos casos de teste a partir do modelo estrutural da aplicação web testada. Sua estrutura básica de funcionamento é uma extensão da ferramenta *GUITAR* (NGUYEN et al., 2014) para aplicações web. Segundo os estudos de Aho et al. (2011), a *GUITAR* é uma das ferramentas mais avançadas de código aberto

[1] http://www.cs.umd.edu/∼atif/
[2] http://www.cs.umd.edu/∼atif/GUITAR−Web/download.htm
[3] http://www.qatestingtools.com/sourceforge/androidGuitar
[4] http://www.qatestingtools.com/testing−tool/iphone_guitar
[5] http://sourceforge.net/apps/mediawiki/guitar/index.php?title=webguitar

disponível para *download* e permite criar automaticamente modelos estruturais para TBM com base na execução e engenharia reversa de uma aplicação existente.

A *GUITAR* é composta por quatro módulos (NGUYEN et al., 2014):

1. ***Ripper*** – módulo responsável pela geração da árvore GUI (modelo estrutural GUI) por meio da execução da aplicação testada;
2. **Conversor de grafo** – módulo responsável pela conversão da árvore GUI em um grafo, denominado Grafo de Fluxo de Evento (EFG);
3. **Gerador de dados de teste** – módulo responsável pela geração automática de casos de teste; e
4. ***Replayer*** – módulo responsável pela execução automática dos casos de teste gerados.

Na ferramenta *WG* os módulos *Ripper* e *Replayer* foram alterados e estendidos para aplicações web. Enquanto o Conversor de grafo e o Gerador de casos de teste são independentes da plataforma, o *Ripper* e o *Replayer* interagem diretamente com a aplicação em teste, portanto exigem adaptações para a plataforma específica. O fluxo de trabalho da ferramenta *WG* é apresentado na Figura 4.9.

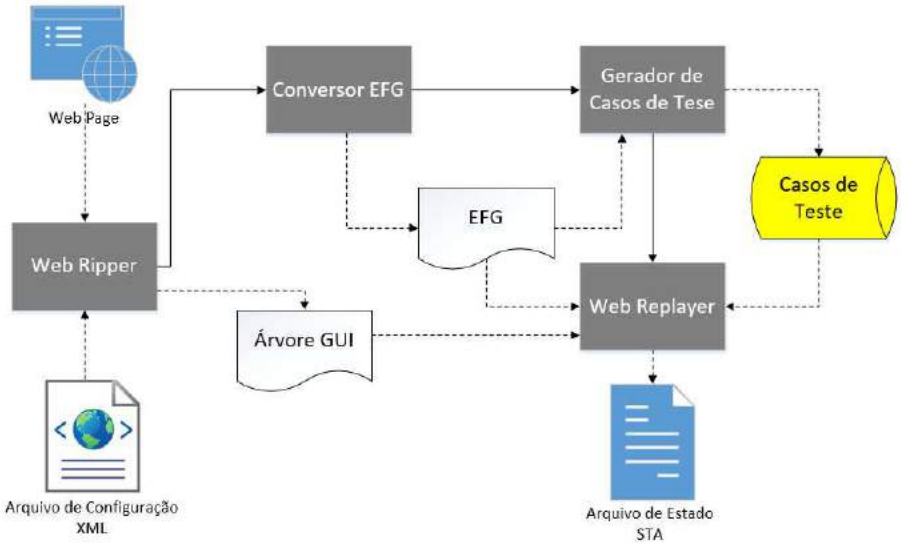


Figura 4.9: *Fluxo de Trabalho da Ferramenta Web GUITAR (adaptado de Nguyen et al. (2014)).*

Ao terminar de explorar a aplicação, o módulo "*Web Ripper*" gera o modelo estrutural (chamado de árvore GUI) como artefato e também permite gerar o mapa do site, que mostra a navegação entre as páginas da aplicação. A árvore GUI refere-se ao modelo estrutural da aplicação que contém o estado de todas as páginas executadas pelo "*Web Ripper*" e trata-se de um arquivo no formato *XML*. O módulo de "Conversor de

EFG" utiliza a árvore GUI para gerar o "Grafo de Fluxo de Evento" (EFG), que possui todos os eventos identificados e as interações entre eles.

O EFG é estruturado como um arquivo *XML*, que serve como entrada para que o módulo "Gerador de Casos de teste". Esses casos de teste são sequências de eventos, sendo que cada nó do grafo representa um evento e todos os eventos que podem ser executados imediatamente após um determinado evento possuem uma aresta partindo do mesmo. Durante a geração de casos de teste, há uma redução dos casos de teste duplicados e/ou não alcançáveis. Porém, nenhum técnica de otimização é utilizada para geração, sendo gerados apenas percorrendo-se o EFG em profundidade.

O "*Web Replayer*" reúne informações da árvore GUI, do EFG e de um caso de teste para a executar o caso de teste definido, partindo do mesmo estado inicial utilizado pelo *Web Ripper*. Para cada evento no caso de teste, o "*Web Replayer*" usa as informações contidas na árvore GUI e do EFG para identificar as páginas e os componentes que o evento precisa para ser executado. Ao final deste processo, é gerado um arquivo de estado (*State File* – STA) que contem o estado da aplicação web após cada etapa intermediária do caso de teste, em que o estado da aplicação é a coleção de páginas atuais e seus componentes correspondentes.

Na ferramenta *WG*, os componentes terminais são definidos como os botões que geralmente estão no final da página, como por exemplo, os botões *Save* e *Cancel*. O conceito de "largura" na ferramenta refere-se ao número máximo de *links* que podem ser explorados em uma única página. Já o conceito de "profundidade" refere-se à quantidade máxima de níveis que podem ser alcançados a partir da URL informada, ou seja, páginas que estão dentro de páginas. Esses parâmetros podem ser configurados no "Arquivo de Configuração" que também se trata de um arquivo *XML*.

Em alguns trabalhos são identificadas limitações da ferramenta *GUITAR* que foram herdadas pela ferramenta *WG*. No trabalho de Aho et al. (2011) são relatadas dificuldades em usar a ferramenta *GUITAR* para gerar modelos de aplicações JAVA com GUI complexas. Mariani et al. (2012) ressaltam problemas da ferramenta *GUITAR* ao interagir com algumas janelas frequentemente utilizadas, como janela de *login*. Nguyen et al. (2014) descrevem várias restrições da ferramenta *GUITAR*, todas relacionadas ao módulo *Ripper*, como por exemplo: execução do *Ripper* dentro de uma única instância da aplicação, levando à geração de árvores GUI imprecisas; a árvore GUI gerada pelo *Ripper* requer validação manual; e problemas na identificação de componentes GUI durante a exploração da aplicação. Esses problemas influenciam diretamente na precisão das árvores GUI e na quantidade de dados de teste gerados e executados. O trabalho de Meireles (2015) propõe melhorias em algumas dessas limitações, resultando numa versão da *GUITAR* denominada *Web GUITAR Modificada*, que neste trabalho irá ser referenciada

como *WG-Modificada*. Basicamente, foram trabalhados a evolução do "Arquivo de Configurações" e dos módulos *Web Ripper* e *Web Replayer*. Destaca-se as seguintes alterações:

- Modificação nos valores de entrada – o "Arquivo de Configuração" na versão original do *Web Ripper* deveria receber os componentes terminais e páginas ignoradas. Ao verificar que esse arquivo também poderia receber valores de campos de entrada, o *Web Ripper* foi modificado para que ele gerasse o arquivo de configuração com as páginas percorridas e os campos de entradas dessas páginas;
- Inclusão do conceito de várias instâncias – como o *Web Ripper* original trabalha apenas com uma instância para cada página visitada, foi necessário incluir o conceito de várias instâncias. Esse modelo permite que uma página possa ser utilizada em suas possíveis variações de estados. Por exemplo, mudança dos campos a serem preenchidos em um determinado formulário em tempo de execução; e
- Inclusão de número máximo de visitas – que se refere a quantidade de vezes que uma página poderá ser explorada e é necessária para se estabelecer uma condição de parada, caso o *Web Ripper* não encontre o número de instâncias desejado.

Porém, Meireles (2015) destaca algumas limitações que foram observadas no decorrer da sua pesquisa:

- Problema na identificação de componentes gerados dinamicamente ou que não possuam identificação (ID e nome da *tag*);
- A não verificação e controle de dependência entre objetos, como em um controle de autenticação no qual o *login* e senha são dependentes;
- Problemas com algumas funções *JavaScript* em que a ferramenta identifica essas funções, mas não permite acessar a página a partir da URL gerada (URL+função *JavaScript*);
- Problema de preenchimento de dados de entrada que torna a execução da ferramenta mais demorada, conforme se aumenta a quantidade de componentes de entrada da página;
- A forma atual da configuração dos parâmetros iniciais para execução do *Web Ripper* é feita manualmente e, com isso, gasta-se muito tempo executando-se essa tarefa, principalmente se o testador não possui conhecimento da aplicação testada; e

Perante as limitações apresentadas, buscou-se na literatura e com especialistas conhecer outras ferramentas e/ou técnicas para para aplicação dos estudos propostos neste trabalho. Na Seção 4.4.3 é apresentada uma nova abordagem neste contexto.

4.4.3 PBGT – *Pattern-Based Gui Testing*

A ferramenta da equipe do professor Memon se utiliza da engenharia reversa a partir da GUI para construção de modelos, como apresentado anteriormente na Seção 4.4.2. Inicialmente esta ferramenta cria o que o autor chama de Árvore GUI (*GUI Forest*), que é uma representação da estrutura das janelas (nós) e a sua relação hierárquica. Cada nó desta estrutura apresenta os seus componentes e respectivas propriedades e valores. Dentro desse contexto das pesquisas de Memon, (PIMENTA, 2007) começou seus trabalhos desenvolvendo um *add-on* da ferramenta *Spec Explorer* (VEANES et al., 2008). O processo começa pela modelagem da GUI com recurso à linguagem *Spec#* (BARNETT; LEINO; SCHULTE, 2005), com métodos que modelam as ações do utilizador e variáveis de estado que modelam o estado da GUI. A partir deste modelo utiliza-se a *Spec Explorer* para a geração de um conjunto de casos de teste. Essa geração faz-se por meio de dois passos: primeiro é gerada uma máquina de estados finita, pela exploração dos estados; e, em seguida, casos de teste que cumpram o critério de cobertura escolhido, são gerados a partir da máquina do passo anterior.

Para tornar possível a automatização da execução de testes, é necessária a criação de código que simule as ações do utilizador e é então que entra em ação a ferramenta desenvolvida. Esta ferramenta gera este código automaticamente com base num mapeamento efetuado pelo testador entre as ações do modelo e os controles da GUI no qual as ações ocorrem. O mapeamento é feito por meio de um *front-end* da ferramenta, denominado *GUI Spy Tool*, a partir do qual se seleciona a ação do modelo que se quer mapear e depois se escolhe o objeto físico da interface a que a ação desejada será aplicada. Com o código de mapeamento gerado, basta depois compilá-lo como uma biblioteca e referenciá-la no projeto do *Spec Explorer*. Os testes podem então ser executados automaticamente, sem intervenção do testador, e ao final tem-se o relatório de erros gerado comparando o resultado do teste com o resultado esperado pelo modelo. Sendo uma abordagem que permite descobrir diversos tipos de erros, esse trabalho apresenta um problema, segundo Pimenta (2007), que é o esforço demasiadamente grande para a construção do modelo em *Spec#*.

Outra abordagem pelos mesmos autores é o projeto "*Pattern-Based GUI Testing* – PBGT" (MOREIRA; PAIVA, 2014b) que também vai ao encontro da proposta da equipe do professor Memon, inclusive ele também tem trabalhado junto ao grupo de pesquisa PBGT (MOREIRA; PAIVA; MEMON, 2013), e se trata de um projeto criado pela Faculdade de Engenharia da Universidade do Porto (FEUP), com colaboração da Universidade do Minho (UM) e a empresa *TelBit*. Esse projeto é aprovado e financiado pela Fundação para a Ciência e Tecnologia (FCT) e visa melhorar os métodos de teste de interfaces gráficas, contribuindo para a construção de uma ferramenta que permita

automatizar a criação e execução de casos de teste sobre GUIs, por meio da técnica de teste baseado em modelos.

Esse projeto permite obter um modelo a partir da própria GUI via engenharia reversa e a alteração do modelo num nível elevado de abstração por meio de um ambiente de modelagem e de uma linguagem específica do domínio (DSL – *Domain-Specific Languages*) com base nos padrões de comportamento da própria GUI, procurando, assim, minimizar o esforço de construção do modelo. Esse modelo permite o mapeamento entre as ações descritas no modelo e os objetos físicos (botões, caixas de texto, entre outros) da GUI a ser testada. É baseada numa linguagem, chamada de *PARADIGMA – Modeling Environment* (MOREIRA; PAIVA, 2014a), que tem por objetivo simplificar e diminuir o esforço necessário no processo de modelagem, promovendo a reutilização. Assim, os casos de teste são gerados automaticamente a partir dos modelos *PARADIGM*.

A ferramenta está disponível gratuitamente como um *plugin* do *Eclipse*, desenvolvido em cima do *Eclipse Modeling Framework* e seus módulos são (MOREIRA; PAIVA, 2014b):

- ***PARADIGMA*** – linguagem específica de domínio (DSL) para a construção de modelos de teste GUI baseada em padrões de teste de interface do usuário;
- ***PARADIGMA-RE*** – componente de engenharia reversa, cuja finalidade é extrair modelos de páginas web;
- ***PARADIGMA-TG*** – componente automatizado de geração de casos de teste a partir dos modelos do *PARADIGMA*;
- ***PARADIGMA-TE*** – componente de execução dos casos de teste que analisa a sua cobertura e gera relatórios para análise; e
- ***PARADIGMA-ME*** – ambiente de modelagem que suporta o construção e configuração de modelos de teste.

O processo *PBGT* define um total de seis etapas principais: modelagem, configuração, geração de casos de teste, execução de casos de teste, análise de resultados e atualização do modelo (quando necessário). Segundo Moreira e Paiva (2014b), tanto a geração de casos de teste quanto a execução são passos totalmente automatizados. A fase de modelagem pode ser realizada manualmente a partir do zero (em um processo de desenvolvimento de software para a frente) ou pode ser executado automaticamente, para obter parte do modelo por um processo de engenharia reversa de um aplicativo de software existente em teste. Isso é feito pelo componente PARADIGMA-RE, que explora o aplicativo em teste e infere o conjunto de padrões de interface do usuário edifício existente, depois, um modelo PARADIGMA com os padrões de teste de interface do usuário que são apropriados para testá-los.

A *PBGT* exige um modelo da GUI escrito na linguagem *PARADIGMA* que tem como objetivo reunir abstrações de domínio aplicáveis, ou seja, padrões de teste de

interface do usuário, permitindo especificar as relações entre eles e também fornecer uma maneira de estruturar modelos em diferentes níveis de abstração, a fim de lidar com a complexidade. A *PARADIGMA* foi construída tendo a reutilização como uma de suas forças, permitindo que "Padrões de Teste de UI" possam ser adaptados de uma aplicação para outra.

A desvantagem está por conta da grande intervenção humana na construção do modelo, com a linguagem *PARADIGMA*, além da necessidade de alguns dados de teste serem criados manualmente no ambiente de modelagem. Outra desvantagem significativa é a falta de suporte para *Patterns* – Padrões, isto é, existem ainda muitos elementos na maioria das interfaces que não suportam os *Patterns* que deveriam para executarem determinadas ações.

4.5 Considerações Finais

Neste capítulo, foi aplicado um mapeamento sistemático para identificação do estado da arte no contexto de teste GUI. Este estudo foi referente ao anos de 1998 a 2012 e selecionou 39 trabalhos entre conferências regulares e de alto nível, que corresponde a 6,52% do total de trabalhos identificados na primeira busca. Verificou-se que esse percentual se justifica devido a duas razões: (1) há muitas obras que se aplicam, avaliam e propõem técnicas para gerar dados de teste, mas não usando como referência a GUI; e (2) os trabalhos se concentram na geração de dados de teste para usabilidade da GUI e não a utilizam como artefato para geração de dados de teste com o objetivo maximizar a cobertura de código-fonte. Perante esse cenário, alguns trabalhos foram estudados e as principais ferramentas e técnicas utilizadas foram detalhadas, como a *Web GUITAR* e a *PBGT*, pontuando os problemas e limitações com suas aplicações. De encontro a esses problemas e limitações, este trabalho propõe, no Capítulo 5, uma proposta de automatização na geração de dados de teste junto a ferramenta *Web GUITAR* e, no Capítulo 6, é definido um meta-modelo para representação de WUIs, procurando dar subsídios para automação da atividade de teste de software.

CAPÍTULO 5

Definição e Aplicação de um Algoritmo Genético junto a Ferramenta *Web GUITAR*

5.1 Considerações Iniciais

A computação inspirada na natureza atrai muita atenção. A natureza serve como uma rica fonte de conceitos, princípios e mecanismos para o projeto de sistemas computacionais artificiais. Entre esses sistemas estão os algoritmos evolucionários, como exemplo: Programação Genética (PG), Programação Evolucionária e os Algoritmos Genéticos (AGs).

O interesse dos pesquisadores da área de computação em biologia é consequência do bom desempenho de estruturas biológicas na resolução de problemas difíceis, inerentes à vida e à sobrevivência. Para alguns biólogos, um dos mecanismos que leva a estas proezas notáveis de solução de problemas é a seleção natural (Charles Darwin). Dada a grande quantidade de problemas difíceis, os pesquisadores da área de computação aplicam os bem sucedidos mecanismos encontrados na natureza para solucionar esses problemas.

Como exemplos de problemas de otimização de difícil solução, o trabalho de Michalewicz (1996) pontua: otimização de funções matemáticas, otimização combinatória, otimização de planejamento, problemas de roteirização, otimização de leiaute de circuitos, otimização de distribuição e de negócios. Observa-se que grande parte desses problemas modelam aplicações reais. Credita-se a grande aplicabilidade dos AGs ao bom desempenho e à fácil adaptação aos problemas, dada a estrutura evolutiva básica e modular desenvolvida.

Desenvolvido por John Holland e popularizado por Goldberg (1989), os AGs consistem em métodos de busca e otimização inspirados em princípios da genética de G. Mendel e na teoria da evolução natural das espécies de Darwin Haupt e Haupt (2004). Segundo Darwin, os indivíduos mais aptos têm, em condições iguais de ambiente, maior chance de reproduzirem e, assim, terem mais descendentes, e propagarem seus códigos genéticos (cromossomos) para as próximas gerações. Portanto, pode-se afirmar que as

boas informações genéticas perpetuam ao longo do tempo, ajudando o melhoramento da espécie.

A aplicação desses conceitos na área de teste de software tem culminado em uma área de pesquisa, dentro da SBSE – *Search-based Software Engineering*, denominada SBST – *Search-Based Software Testing*, que foca a aplicação de técnicas de otimização matemática na resolução de problemas encontrados com a aplicação da atividade de teste de software. Dentro desse contexto, Harman (2006) afirma que, entre as técnicas de otimização aplicadas na literatura, se destaca o uso do AG, sendo aplicado em, aproximadamente, 80% dos trabalhos. O mapeamento sistemático, descrito no Capítulo 4 dessa tese, também reforça essa informação. Isso se justifica por essa técnica ser de fácil adaptação aos problemas inerentes da atividade de teste de software. Sendo assim, optou-se por utilizá-lo na tentativa de gerar conjuntos de dados de teste melhores do que os gerados pela ferramentas de teste UI. Por esse motivo será detalhado a seguir.

5.2 Conceitos Básicos sobre AG

Análogos à natureza, os AGs evoluem o código genético da população durante as gerações para adaptar-se e resolver um problema específico, mesmo sem ter informações detalhadas do problema. Assim, pode-se afirmar que os AGs, na busca por soluções para o problema, empregam um processo adaptativo, já que a informação corrente influencia a busca futura, e paralelo, pois várias soluções são consideradas ao mesmo tempo. O trabalho do Camilo (2010) faz uma analogia entre os AGs e a natureza, que é apresentada na Tabela 5.1.

Tabela 5.1: *Termologias Usadas Pelos AGs – Analogia com a Natureza.*

Natureza	**Algoritmo Genético**
Cromossomo	Estrutura de dados que representa a solução.
Indivíduo	Mesmo que cromossomo.
Gene	Característica (variável que compõe o cromossomo).
Alelo	Valor da característica.
Locus	Posição do gene no cromossomo.
Genótipo	Cromossomo codificado.
Fenótipo	Cromossomo decodificado.
Populacão	Conjunto de soluções.
Geracão	Ciclo.

Esses algoritmos simulam processos naturais de sobrevivência e reprodução das populações, essenciais em sua evolução. Na natureza, indivíduos de uma mesma população competem entre si, buscando principalmente a sobrevivência, seja por meio da busca de recursos como alimento ou visando à reprodução. Os indivíduos mais aptos terão um maior número de descendentes, ao contrário dos indivíduos menos aptos.

A ideia básica de funcionamento dos algoritmos genéticos é a de tratar as possíveis soluções do problema como "indivíduos" de uma "população", que irá "evoluir" a cada iteração ou "geração". Para isso, é necessário construir um modelo de evolução no qual os indivíduos sejam soluções de um problema. A execução do algoritmo pode ser resumida nos seguintes passos e é representada pela Figura 5.1:

1. Inicialmente, escolhe-se uma população inicial, normalmente formada por indivíduos criados aleatoriamente;
2. Avalia-se toda a população de indivíduos segundo algum critério (avaliação de aptidão), determinado por uma função que avalia a qualidade do indivíduo (função de aptidão ou "*fitness*");
3. Em seguida, por meio do operador de "seleção", são escolhidos os indivíduos de melhor valor (dado pela função de aptidão) como base para a criação de um novo conjunto de possíveis soluções, chamado de nova "geração";
4. Esta nova geração é obtida aplicando-se sobre os indivíduos selecionados operações que misturem suas características ("genes"), utilizando operadores de cruzamento (*crossover*) e de mutação;
5. Estes passos são repetidos até que uma solução aceitável seja encontrada ou até que o número predeterminado de passos seja atingido, formando um fluxo cíclico.

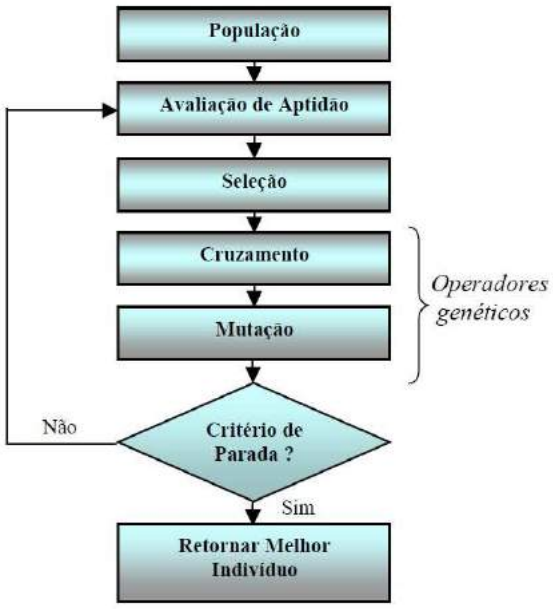


Figura 5.1: *Estrutura Básica de um AG (adaptado de Goldberg (1989)).*

Segundo Goldberg (1989), esses algoritmos diferem-se dos algoritmos tradicionais de otimização em basicamente quatro aspectos:

1. Baseiam-se em uma codificação do conjunto das soluções possíveis, e não nos parâmetros da otimização em si;
2. Não necessitam de nenhum conhecimento derivado do problema, apenas de uma forma de avaliação do resultado;
3. Os resultados são apresentados como uma população de soluções e não como uma solução única; e
4. Usam transições probabilísticas e não regras determinísticas.

O Algoritmo 5.1 mostra um trecho de pseudo-código descrevendo um algoritmo genético adaptado do trabalho de Goldberg (1989).

Algoritmo 5.1 Exemplo de Algoritmo Genético.

```
1 Entrada população -> uma lista de indivíduos
2 Entrada função-objetivo -> recebe um indivíduo e retorna um número real
3 Saída indivíduo
4
5 enquanto nenhuma condição de parada for atingida faça
6   lista de pais = seleção(população, função-objetivo);
7   população = reprodução(lista de pais);
8 fimenquanto
9 retorne o melhor indivíduo da população de acordo com a função-objetivo
```

Segundo Goldberg (1989), a função-objetivo é o alvo da otimização e, pode ser um problema de otimização, um conjunto de teste para identificar os indivíduos mais aptos, ou mesmo uma "caixa preta" na qual se sabe apenas o formato das entradas e nos retorna um valor que se queira otimizar. A grande vantagem dos AGs está no fato de não precisar saber como funciona esta função objetivo, apenas tê-la disponível para ser aplicada aos indivíduos e comparar os resultados.

5.2.1 Indivíduo e População

No AG o indivíduo é uma representação do espaço de busca do problema a ser resolvido, em geral, representado na forma de sequências de *bits* ou caracteres. O indivíduo é meramente um portador do seu código genético. O código genético é uma representação do espaço de busca do problema a ser resolvido, muitas vezes, na forma de sequências de *bits*. Por exemplo, para otimizações em problemas cujos valores de entrada são inteiros positivos de valor menor que 255 pode-se utilizar 8 *bits*, com a representação binária normal. Nem sempre é fácil a tarefa do mapeamento do espaço de possíveis soluções para o problema, podendo haver a necessidade de envolver heurísticas adicionais como codificadores (FERREIRA; VERGILIO, 2004).

A população de um algoritmo genético é o conjunto de indivíduos que estão sendo cogitados como solução e que serão usados para criar o novo conjunto de indivíduos para análise. A população inicial é o ponto de partida dos AGs, por isso, tem grande

influência na convergência e no desempenho do algoritmo (TOUGAN; DALOUGLU, 2008). Uma boa população inicial deve apresentar diversidade, para dar aos AGs o maior número de matérias-prima possível. Normalmente a geração da população inicial é feita de maneira aleatória, apesar de existirem outras abordagens que podem ser encontradas no trabalho de Camilo (2010). Um parâmetro de extrema importância para a execução do AG é o tamanho da população. Segundo Tougan e Dalouglu (2008), quando uma população é pequena, pode ocorrer uma estagnação no processo evolutivo, dependendo da evolução e da pouca variabilidade genética. Já uma população grande demais, poderá tornar o algoritmo extremamente lento e com baixo rendimento em termos de processamento computacional.

5.2.2 Avaliação de Aptidão (*Fitness*) e Seleção

A função de *fitness* (f) pode ser pensada como uma medida de desempenho, lucratividade, utilidade e excelência que se queira maximizar (GOLDBERG, 1989). O *fitness* é associado ao grau de resistência e adaptabilidade ao meio no qual o indivíduo vive. É por meio desta função que se mede quão próximo um indivíduo está da solução desejada ou quão boa é esta solução. Com isso, indivíduos com maior f terão maior chance de sobreviver e serão responsáveis pela próxima geração. Nos AGs é associado um valor numérico de adaptação, o qual se supõe que é proporcional à sua "utilidade" ou "habilidade" em função do seu objetivo. É essencial que esta função seja muito representativa e diferencie, na proporção correta, as boas das más soluções. Se houver pouca precisão na avaliação, uma ótima solução pode ser posta de lado durante a execução do algoritmo, além de gastar mais tempo explorando soluções pouco promissoras (TOUGAN; DALOUGLU, 2008).

A seleção é feita após o cálculo da aptidão, no final são escolhidos alguns indivíduos para serem aplicados os operadores genéticos. A seleção considera o valor do *fitness*, ou seja, o indivíduo com maior *fitness* tem maior probabilidade de formar descendentes melhor adaptados. Geralmente os Algoritmos Genéticos utilizam um método de seleção para obter um par de indivíduos que se tornarão os pais de dois novos indivíduos para a nova geração (FERREIRA; VERGILIO, 2004). A seleção também é considerada outra parte chave do algoritmo e pode-se usar o algoritmo de seleção por Roleta (*Rolet*), no qual os indivíduos são ordenados de acordo com a função-objetivo e lhes são atribuídas probabilidades decrescentes de serem escolhidas – probabilidades essas proporcionais à razão entre a adequação do indivíduo e a soma das adequações de todos os indivíduos da população. A escolha é feita então aleatoriamente de acordo com essas probabilidades. Dessa forma conseguimos escolher como pais os mais bem adaptados, sem deixar de lado a diversidade dos menos adaptados.

Além do método da Roleta, existem outros métodos que podem ser implementados para se efetuar a seleção. Blickle e Thiele (1996) descrevem alguns desses métodos em seu trabalho:

- Integral – respeita rigidamente o *fitness* relativo;
- Aleatória – são selecionados aleatoriamente N indivíduos da população;
- *Ranking* – indivíduos são ordenados de acordo com seu escore de *fitness* e a chance de que cada indivíduo ser selecionado é dada pela posição do *ranking*, não mais pelo valor absoluto do *fitness*. Este método pode resultar em convergência acelerada e manter a variância da probabilidade de seleção dos indivíduos de cada população a mesma;
- Torneio – indivíduos selecionados aleatoriamente disputam um torneio, no qual o melhor é selecionado para a reprodução. Este método é mais eficiente computacionalmente que o ranking e também diminui a probabilidade de convergência acelerada. Este método é o mais utilizado, pois oferece a vantagem de não exigir que a comparação seja feita entre todos os indivíduos da população (TOMMASEK, 2012); e
- Elitismo – força-se o algoritmo genético a reter o melhor indivíduo da geração atual, ou alguns dos melhores, na próxima geração. Aplicado, geralmente, como complemento do método da Roleta, pois como a probabilidade de ser escolhido não é 100%, há a possibilidade de que o melhor indivíduo não seja escolhido para a próxima geração, preservando os melhores *fitness* para a próxima geração.

5.2.3 Operadores Genéticos

O principio básico dos operadores genéticos é transformar a população por meio de sucessivas gerações, estendendo a busca até chegar a um resultado satisfatório. Os operadores genéticos são necessários para que a população se diversifique e mantenha características de adaptação adquiridas pelas gerações anteriores. Os operadores de cruzamento e de mutação têm um papel fundamental em um algoritmo genético.

5.2.3.1 Cruzamento (*Crossover*)

Também chamada de recombinação ou *crossing-over* é um processo que imita o processo biológico homônimo na reprodução sexuada, ou seja, os descendentes recebem em seu código genético parte do código genético do pai e parte do código da mãe. Essa recombinação garante que os melhores indivíduos sejam capazes de trocar entre si as informações que os levam a ser mais aptos a sobreviver, e assim gerar descendentes ainda mais aptos.

Segundo Goldberg (1989), os tipos de cruzamento mais utilizados são o cruzamento em um ponto e o cruzamento em dois pontos, mostrados nas Figuras 5.2 e 5.3, respectivamente. Com um ponto de cruzamento, seleciona-se aleatoriamente um ponto de corte do cromossomo. Cada um dos dois descendentes recebe informação genética de cada um dos pais. Com dois pontos de cruzamento, um dos descendentes fica com a parte central de um dos pais e as partes extremas do outro pai e vice versa.

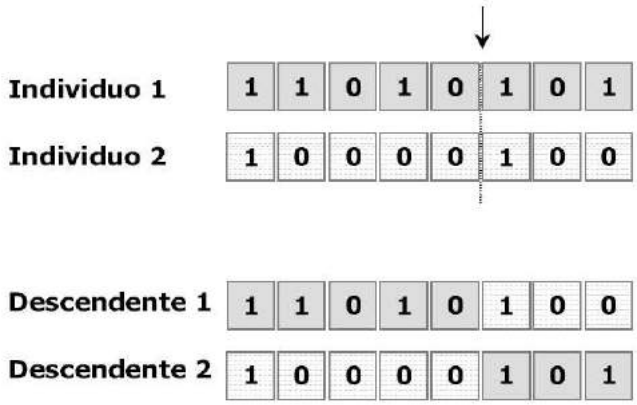


Figura 5.2: *Cruzamento em um ponto (adaptado de Camilo (2010).*

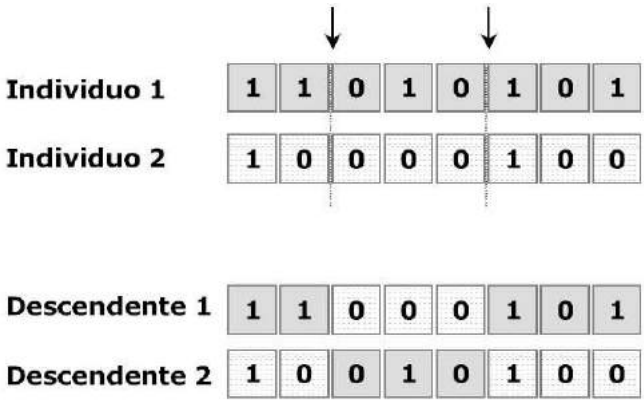


Figura 5.3: *Cruzamento em dois pontos (adaptado de Camilo (2010).*

Existem outro métodos de cruzamento, como o cruzamento uniforme, no qual todos os alelos são trocados com uma certa probabilidade p_e que é conhecida como probabilidade de troca (*swapping probability*). Porém, os métodos de cruzamento descritos nessa seção foram os analisados para serem implementados no AG proposto neste trabalho e que é apresentado na Seção 5.3.

5.2.3.2 Mutação

Esta operação simplesmente modifica aleatoriamente alguma característica do indivíduo sobre a qual é aplicada, conforme ilustrado na Figura 5.4. Essa troca é importante, pois propicia a inserção de novos valores de características que não existiam ou apareciam em pequena quantidade na população em análise. O operador de mutação é necessário para a introdução e manutenção da diversidade genética da população. Dessa forma, a mutação assegura que a probabilidade de se chegar a qualquer ponto do espaço de busca possivelmente não será zero, permitindo maior variabilidade genética na população e impedindo que a busca fique estagnada em um mínimo local (GOLDBERG, 1989; CAMILO, 2010). O operador de mutação é aplicado aos indivíduos por meio de uma taxa de mutação a ser definir e que, geralmente, é pequena.

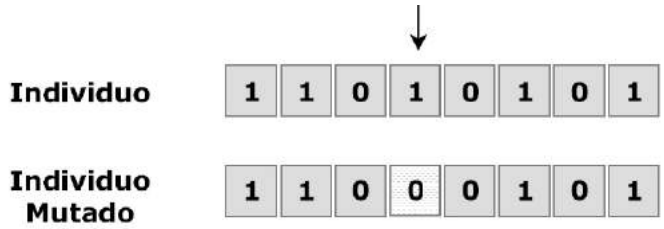


Figura 5.4: *Mutação Simples.*

Entre os operadores de mutação para codificação binária existe a mutação chamada de Clássica, na qual o operador de mutação varre todos os genes do cromossomo e a cada gene muta-se o valor com a probabilidade p_m (taxa de mutação). A mutação é feita pela inversão do gene, onde for 0 altera-se para 1 e onde for 1 altera-se para 0. Deve-se procurar um valor de p_m que permita um balanço entre a descoberta de novas soluções e, ao mesmo tempo, que não provoque excessiva destruição dos bons blocos de material genéticos já descobertos. Sugere-se que p_m seja $1/t$, onde t é o número de genes do cromossomo (BäCK, 1993; GOLDBERG; VOESSNER, 1999).

Assim como os operadores de cruzamento, os operadores de mutação também são dependentes da codificação. Portanto, além do operador citado, que é utilizado na codificação binária, existem outros operadores de mutação e que são detalhados no trabalho de Larranaga et al. (1999).

Ao elaborar um AG, deve-se considerar alguns parâmetros que influenciam no sucesso de seu funcionamento. Um desses parâmetros é o tamanho da população. Esse tamanho influência na exploração do espaço de busca por possíveis soluções, no tempo de execução e na demanda por recursos computacionais. Uma população pequena pode significar uma amostragem insuficiente do espaço de busca. Uma população grande pode levar uma convergência mais lenta, exigindo mais recursos computacionais ou aumento do tempo necessário para execução do algoritmo. A taxa de cruzamento, que controla

a frequência com a qual o operador de cruzamento é aplicado, caso seja baixo, pode significar pouco aproveitamento da informações existente, já um um alto valor pode provocar convergência prematura. O mesmo acontece com a taxa de mutação, que define a probabilidade de um indivíduo ter seus genes alterados. Se o valor for muito baixo, pode não satisfazer a necessidade de exploração e levar o algoritmo à estagnação. Porém, um alto valor dessa taxa pode conduzir a uma busca aleatória.

Um dos principais parâmetros é o critério de parada, que determina quando a execução do AG irá parar. Existem várias formas de determinar essa taxa. Entre elas, pode-se utilizar como um valor conhecido (meta), o número de chamadas à função de avaliação e o número de gerações, sendo as duas últimas as mais usadas na literatura.

A escolha ideal dos parâmetros é um problema não linear e depende do tipo de problema tratado. Por isso, não é possível encontrar uma boa configuração para generalizar a execução de qualquer tipo de problema (CAMILO, 2010).

Os AGs diferem-se dos métodos tradicionais determinísticos de otimização, pois utilizam meios probabilísticos para encontrar o resultado, trabalham com parâmetros codificados e avaliam cada indivíduo isoladamente, possuindo um paralelismo implícito. Algumas das vantagens de se utilizar AG são (HAUPT; HAUPT, 2004):

- A capacidade de otimizar variáveis contínuas ou discretas;
- O fato de não requerer informações de derivadas;
- A habilidade de efetuarem busca simultânea a partir de um grande número de pontos;
- O fato de lidar com um grande número de variáveis;
- A boa adaptabilidade para ambientes de processamento paralelo;
- A facilidade para fugir de mínimos locais, podendo trabalhar com variáveis codificadas;
- O fato de proporcionar um ótimo conjunto de soluções e não apenas uma única;
- Otimização feita por meio da codificação de variáveis; e
- A propriedade de trabalharem com dados gerados numericamente, dados experimentais ou funções analíticas.

Apesar disso, os AGs não devem ser considerados como a melhor alternativa para resolução de todos os problemas, apesar de serem empregados com sucesso para solucionar uma vasta variedade deles. Isso porque são mais eficientes na resolução de problemas complexos, com grande número de variáveis e espaço de busca não linear. São empregados também com sucesso na resolução de problemas para os quais não há conhecimento prévio sobre a resolução do problema ou quando não se tem conhecimento de qualquer outra técnica específica para solucionar o problema (CAMILO, 2010).

5.3 *GAWG* – AG Aplicado a *Web GUITAR*

Analisando os trabalhos selecionados no mapeamento sistemático do Capítulo 4, que abordam a geração e a seleção de dados de teste usando AG, existe grande semelhança na forma de representação do indivíduo. Nesses, o indivíduo é um conjunto de dados de teste e a população um conjunto de indivíduos. Dessa forma, cada conjunto de dados de teste tem um tamanho pré-fixado na modelagem do algoritmo, o que pode acarretar em um desperdício de recursos computacionais e de tempo, pelo fato que podem existir dados de teste que não sejam úteis ou redundantes. Devido a esses fatores, é proposta uma nova representação genética e uma nova heurística para customizar a geração dos dados de teste. Pretende-se criar dados de teste otimizados, ou seja, um conjunto pequeno de entradas e que tenham um bom desempenho de cobertura. Para isso, a princípio foi proposto um AG adaptado para o fluxo de execução da *WG-Original*, como o nome de *GAWG* (***AG** Applied to **W**eb **GUITAR***).

Inicialmente o *GAWG* foi aplicado junto à ferramenta *WG-Original*, visando evoluir o conjunto de dados de teste. Como configuração inicial, foi feita a substituição do gerador de dados de teste proprietário da ferramenta *WG-Original* pelo *GAWG*, como contextualizado pela Figura 5.5. O *GAWG* precisou ser estruturado para ler como entrada o artefato gerado (arquivo EFG) pela ferramenta *WG-Original*, que antes era interpretado pelo gerador de dados de teste nativo da ferramenta, como anteriormente apresentado na Figura 4.9 (p. 86).

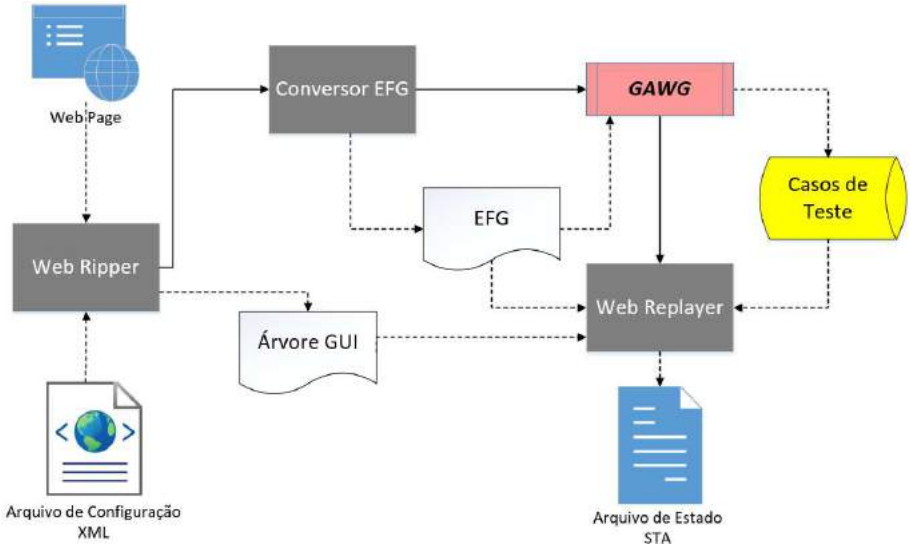


Figura 5.5: *Utilização do GAWG Junto a Ferramenta WG-Original.*

5.3.1 Representação e Fluxo de Execução

A representação utilizada no *GAWG* leva em consideração que cada cromossomo representa um dado de teste no qual cada gene representa um evento a ser executado em um objeto. Ou seja, cada cromossomo é formado de genes que estão relacionados a um *widget*. Esses genes são os eventos associados a cada *widget*, sempre começando o processo por um gene inicial, ou seja, um evento de um *widget* que está acessível pelo usuário inicialmente. Cada gene subsequente é gerado a partir da ligação do gene anterior, utilizando-se uma matriz de adjacência para isso. O conjunto desses genes preenchidos formam um indivíduo, sendo um conjunto de indivíduos uma entrada para o estudo exploratório abordado neste trabalho e, consequentemente, a população representa um conjunto de dados de teste como representado na Figura 5.6.

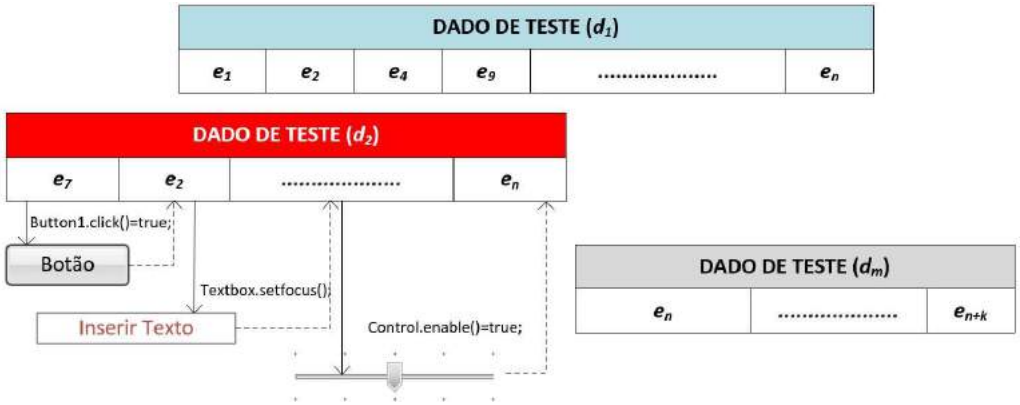


Figura 5.6: *Representação do Conjunto de Dados de Teste Gerado pelo AG.*

Formalizando a representação do conjunto de dado de teste, assume-se que D é um conjunto de dados de teste formado por $D = \{d_1, d_2, ..., d_m\}$, sendo $m \geq 1$. Com isso, define-se um dado de teste d como sendo um par $d = \{e_w, e_1; e_2; ...; e_n\}$, sendo que e_w trata-se de um estado inicial e d uma sequência de eventos executável $e_1; e_2; ...; e_n$. As definições de estado inicial e sequência de eventos executável foram descritas na Seção 3.4.1.

Para aplicação do cruzamento, são trocadas sequências de eventos válidas entre cromossomos, sendo que o tipo de cruzamento escolhido para ser aplicado foi o de um ponto. Para isso, toda a população é percorrida selecionando pares consecutivos de cromossomos. Sorteia-se um gene do primeiro cromossomo de forma aleatória e o procura no segundo cromossomo. Caso o gene seja localizado, troca-se a calda. Ou seja, seleciona-se de forma sequencial, um par de cromossomos, $cromo_x$ e $cromo_w$. Identifica-se o $gene_y \in cromo_x$ e $gene_y \in cromo_w$, e inverte-se o restante da sequência, como mostra a Figura 5.7.

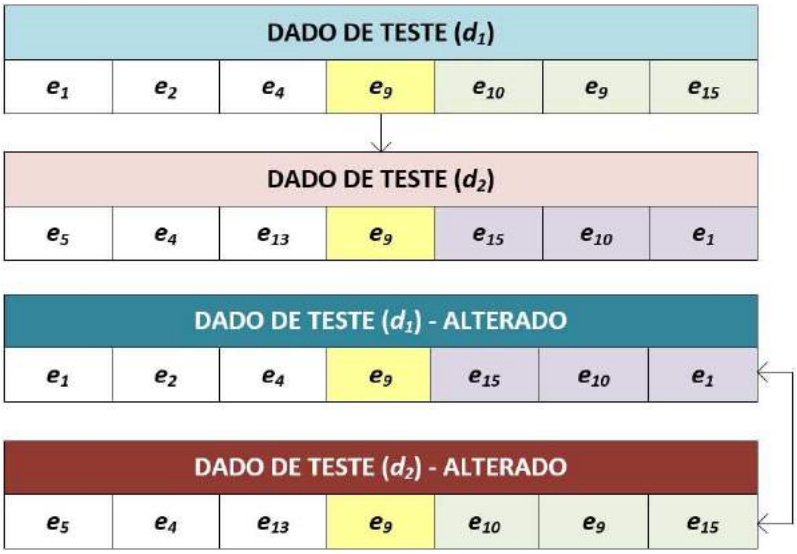


Figura 5.7: *Cruzamento Aplicado pelo GAWG.*

Para aplicação da mutação, é necessário que seja inserida, no cromossomo, uma nova sequência de eventos válida. Isso é necessário porque, se for trocado, por exemplo, um evento e_x por um evento e_y quaisquer, pode ocasionar a geração de uma sequência de eventos não executável. Evitando essa situação, sorteia-se um cromossomo da população de forma aleatória e que possua, no mínimo, três genes. Aleatoriamente identifica-se um gene nesse cromossomo, que precisa possuir irmãos ao seu redor. Selecionam-se os irmãos, tanto da esquerda quanto da direita, para serem procurados, de forma sequencial, no restante da população para se localizar um outro cromossomo que será utilizado para efetuar a mutação. Ou seja, seleciona-se, de forma aleatória, um cromossomo ($cromo_x$). Sorteia-se um $gene_y \in cromo_x$, sendo que $gene_{y-1}$ e $gene_{y+1}$ serão procurados no restante da população e se e somente se forem encontrados em outro cromossomo na mesma sequência de ocorrência, ela é substituída, como representado na Figura 5.8. Nesta figura,o evento e_9 do dado de teste d_1 é substituído pelos eventos e_{13}, e_1 e e_{15} do dado de teste d_2. Observa-se que isso garante uma mutação válida, ou seja, uma sequência de eventos executável, pois os eventos anterior e_4 e posterior e_{10} são preservados.

Essas restrições impostas para o cruzamento e para a mutação foram aplicadas para se evitar a geração de dados de teste que não fossem alcançáveis, ou seja, dados de teste com sequências impossíveis de serem executadas.

A Figura 5.9 representa o fluxo completo de execução do *GAWG*. No primeiro momento é interpretado o arquivo "Arquivo EFG.xml", gerado pela ferramenta *WG-Original*, mapeando-se cada objeto com suas variações. A partir desse mapeamento, é gerada, aleatoriamente, a população inicial. Tanto o tamanho máximo da população quanto dos cromossomos são definidos no arquivo de configuração. Porém, o tamanho de cada cromossomo pode ser variável, sendo sorteado entre 1 e o tamanho máximo.

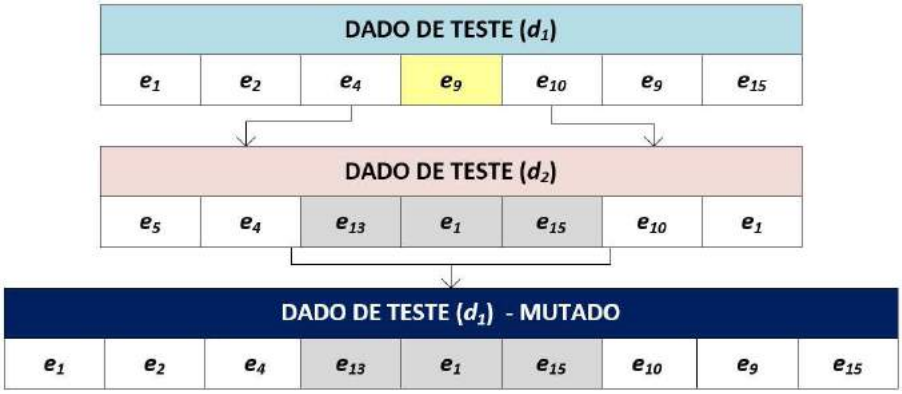


Figura 5.8: *Mutação Aplicada pelo GAWG.*

Tendo uma população inicial gerada, calcula-se a aptidão de acordo com a função objetivo (FO) definida. A FO determina o problema a ser resolvido e, por isso, é o guia do AG durante o processo de otimização. Neste trabalho o objetivo é maximizar a FO, que calcula a cobertura de LOC (linhas de código) para um dado cromossomo, priorizando os de menor tamanho, conforme mostra a Equação 5-1.

$$FO = cobertura(cromossomo, SUT)/tamanho(cromossomo) \tag{5-1}$$

onde, a função *cobertura* verifica qual o percentual de código-fonte foi coberto por determinado cromossomo e a função *tamanho* retorna o quantidade de genes do cromossomo. Indivíduos com maior *fitness* terão maior chances de sobreviver e serão responsáveis pela próxima geração, associando um valor numérico, supondo sua "habilidade" em função do seu objetivo. Nesse contexto, para garantir que o melhoramento seja contínuo, foi utilizada da técnica de elitismo. Segundo Baluja e Caruana (1995), o elitismo deve ser usado quando se quer preservar a melhor solução. Nesse contexto, os melhores cromossomos, no que diz respeito a cobertura de LOC, serão mantidos mesmo que sejam de grande tamanho. Para auxílio ao elitismo, foi desenvolvida uma heurística que identifica n cromossomos pela sua alta cobertura de LOC e já os elegem para a próxima geração. Após isso, selecionam-se os melhores indivíduos de acordo com a FO, ordenando a população de forma decrescente pela cobertura, mas priorizando a relação com seu tamanho. A quantidade dos cromossomos que serão selecionados pelo elitismo (n) é definida no arquivo de configuração.

Para se aplicar o cruzamento e a mutação, são feitas N iterações, que sua quantidade também é definido no arquivo de configuração. Para mutação, aplicou-se a recomendação do trabalho Bäck (1993), que leva em consideração o tamanho do cromossomo. Após a finalização do AG, pode-se determinar os dados de teste eleitos, que

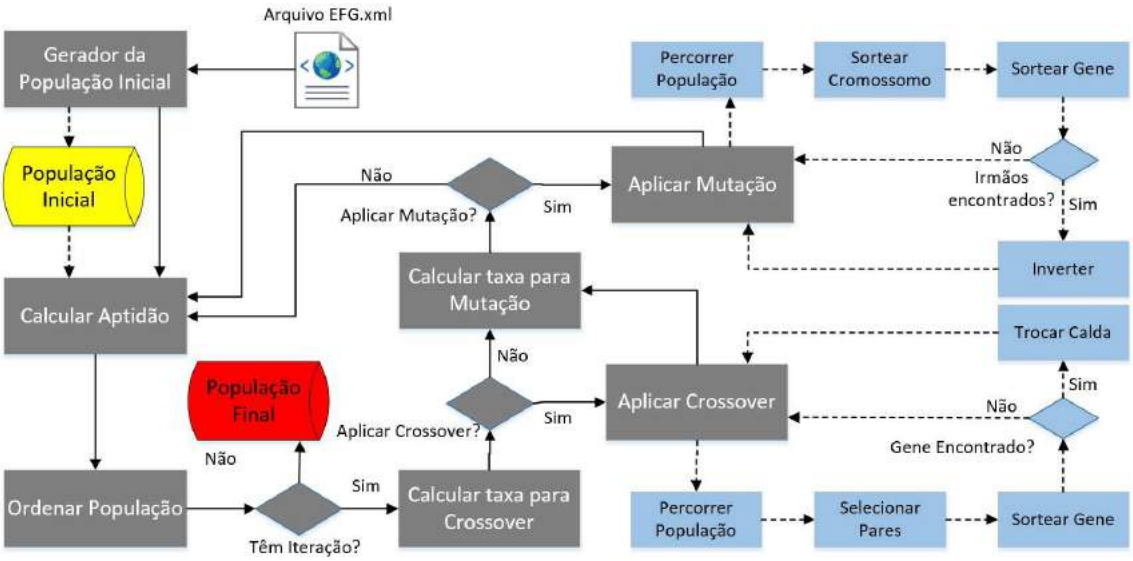


Figura 5.9: *Fluxo de Execução do GAWG.*

correspondem aos indivíduos que atingem uma maior cobertura de código, priorizando os de menor tamanho, considerando que o objetivo é ter uma maior cobertura com o mínimo de esforço computacional.

5.3.2 Estudo Exploratório

Esse estudo tem como objetivo avaliar o algoritmo genético proposto para geração de dados de teste. Para isso, inicialmente utilizou-se a ferramenta *WG-Original* para execução e análise de cobertura dos dados de teste gerados. Sua aplicação é composta das seguintes atividades: seleção de programas, geração de conjuntos de dados de teste, seleção e adaptação das ferramentas de teste, aplicação do estudo exploratório e a coleta e análise dos resultados. A seguir, cada uma dessas atividades é detalhada, considerando as informações sobre o estudo exploratório referente a este trabalho.

5.3.2.1 Seleção das Aplicações Web

Essa atividade é de extrema importância para estudos exploratórios e deve ser realizada com cuidado, tendo-se em mente os objetivos que se deseja atingir (VINCENZI, 1998). Este experimento foi conduzido a partir de um grupo formado por quatro programas utilitários do *TOMCAT* (APACHE, 2016). Tais programas possuem as características necessárias para a aplicação do experimento e foram desenvolvidos utilizando a linguagem de programação JAVA. A Tabela 5.2 apresenta a lista desses programas com a descrição, o número de classes e linhas de código de cada programa.

Tabela 5.2: *Conjunto de Aplicações para o Experimento.*

Programa	*Descrição*	*Número de Classes*	*LOC*
Calendar	Adiciona-se compromisso em um calendário em determina data e horário.	4	112
Carts	Adiciona e remove itens selecionados um a um em uma lista.	2	68
Checkbox	Adiciona itens selecionados simultaneamente em uma lista.	3	74
Colors	Entra com duas cores para alteração de cor da fonte e cor de fundo.	2	65

5.3.2.2 Seleção e Adaptação das Ferramentas de Teste

A seleção das ferramentas de teste deve levar em consideração quais critérios serão analisados, a linguagem nas quais os programas estão escritos e, principalmente, o suporte oferecido para a realização de experimentos, ou seja, deve-se observar se a funcionalidade e o conjunto de informações disponibilizados pela ferramenta são suficientes para atingir os objetivos propostos. Não conseguimos identificar uma única ferramenta que tivesse todas as funcionalidade necessárias para o experimento, que se resumiam em:

1. Fornecer subsídios para construir uma representação de aplicações web que, por meio dela, fosse possível gerar dados de teste;
2. Executar a aplicação web com um conjunto de dados de teste;
3. Instrumentar a aplicação para se calcular a cobertura de LOC; e
4. Gerar relatórios de cobertura de LOC das aplicações em teste.

Com isso, foi decidido utilizar algumas ferramentas em consonância para se obter todas as funcionalidades. Para a funcionalidade do item 1 a ferramenta *WG-Original* foi escolhida inicialmente. Porém, apresentou muitas limitações e, depois de muito esforço na tentativa de utilizá-la, decidiu-se pela adoção da ferramenta *WG-Modificada* (MEIRELES, 2015) que, a princípio, aparentava ter os requisitos para aplicação do experimento e que foi publicada durante o período de tentativa de uso da *WG-Original*. Suas características de funcionamento foram apresentadas na Seção 4.4.2.

Na implementação do estudo exploratório, verificou-se que tanto a *WG-Original* quanto a *WG-Modificada* faziam uso de recursos da ferramenta *Selenium* (DAVID, 2012) para execução dos dados de teste, funcionalidade necessária e apresentada no item 2. Com isso, foi necessário recorrer a instalação dessa ferramenta que se trata de um conjunto de ferramentas: a *Selenium IDE* e o *Selenium WebDriver*.

O *Selenium IDE* (DAVID, 2012) é uma ferramenta que dá subsídio a rápida criação de *scripts* de testes, permitindo gravar, editar, reproduzir e exportar dados de teste (*capture/replay*). Está implementada como uma extensão para o navegador *Mozilla Firefox* e as ações gravadas podem ser executadas várias vezes, podendo serem alteradas manualmente.

Já o *Selenium WebDriver* (AVASARALA, 2014) é uma ferramenta utilizada para gerar testes automáticos de aplicações web e verificar se os resultados estão de

acordo com o esperado. A ferramenta é suportada por vários navegadores sem o uso de *Javascript* e pode ser utilizada em várias linguagens. Possui suporte para navegadores como *Mozilla Firefox*, *Google Chrome*, *Internet Explorer* e *Opera*, além de ambientes *mobiles*. A ferramenta usa uma interface, que é denominada de *Driver*, na qual serão executados todos os passos selecionados pelo programador. Existem diferentes *Drivers* para cada um dos navegadores suportados, por exemplo o $FirefoxDriver()$ para o *Firefox* e o $ChromeDriver()$ para o *Google Chrome*.

Por meio da API (*Application Programming Interface*) do *WebDriver*, é possível executar um variado número de ações relativas ao *Driver*, como abrir o *Driver* no URL indicado, procurar elementos presentes no *Driver*, obter o *HTML* respetivo, entre outras operações. O *Driver* é composto por vários elementos que são denominados $WebElements$. Estes elementos correspondem aos elementos presentes no código *HTML* da página. A pesquisa dos elementos pode ser efetuada por meio de alguns parâmetros: *nome*, *id*, *tagname*, *XPath*, entre outros. Um *WebElement* contém um conjunto de ações pré-definidas que podem ser executadas sobre esse elemento, tal como um clique ou a introdução de texto (por meio do método *SendKeys*). A Figura 5.10 mostra, na linha 1, como se cria um *WebDriver* no navegador *Mozilla Firefox*. Após a criação, é acessada a página da Faculdade de Computação (FACOM) da UFMS (linha 2), da qual é selecionado o primeiro botão (linha 3) e efetuado um clique sobre ele (linha 4).

```
1 driver = new FirefoxDriver(); /*novo driver*/
2 driver.Navigate().GoToUrl(http://facom.sites.ufms.br/) /*acesso a aplicação*/
3 IWebElement OkButton = driver.FindElement(By.TagName("Button"));
4 OkButton.Click() /*clique sobre o botão*/
```

Figura 5.10: *Utilizando Comandos da Ferramenta Selenium WebDriver.*

Para calcular a cobertura de LOC, variável fundamental da função objetivo proposta na Seção 5.3.1, foi utilizada a ferramenta *Cobertura* (COBERTURA, 2016), que se trata de uma ferramenta de software livre, que calcula o percentual de código executado em programas JAVA, identificando quais partes do seu programa ainda não foram cobertas.

5.3.2.3 Coleta dos Dados

Nesta fase, a base de dados a partir da qual as informações serão coletadas foi gerada. Para se coletar os dados necessários, foram desenvolvidos *scripts* para serem aplicados aos relatórios gerados pelas ferramentas. Para aplicação do estudo exploratório, as ferramentas foram devidamente instaladas e geraram muitas dificuldades de configuração. Com as ferramentas devidamente instaladas, determinou-se a seguinte sequência de etapas:

1. Aplicar o módulo *Ripper* da ferramenta *WG-Modificada* para geração do modelo estrutural WUI da SUT;
2. Uma vez que se tem o modelo estrutural, aplicar o módulo *Conversor de grafo* da ferramenta *WG-Modificada* para geração do EFG;
3. Tendo como entrada o EFG, são aplicados o módulo de *Gerador de casos de teste* nativo da ferramenta *WG-Modificada* e, em seguida, o *GAWG* para geração do conjuntos de dados de teste. A Tabela 5.3 mostra o tamanho do conjunto de dados de teste (T) e o tamanho do maior dados de teste (t) gerado pela *WG-Modificada*, comparando com o tamanho do conjunto de dados de teste (W) e o tamanho do maior dados de teste (w) gerado pelo *GAWG*;
4. Instrumentar a aplicação usando a ferramenta *Cobertura*;
5. Para cada dado de teste t, do conjunto de dados de teste T, se faz:
   (a) Executar t utilizando o módulo *Replayer* da ferramenta *WG-Modificada*, utilizando a ferramenta *Selenium WebDriver*;
   (b) Gerar relatório de cobertura, usando a ferramenta *Cobertura*;
   (c) Aplicar o *merge* entre os arquivos de cobertura; e
   (d) Selecionar o próximo dado de teste, ou seja, $t = t + 1$.
6. Para cada dado de teste w, do conjunto de dados de teste W, se faz:
   (a) Executar w utilizando o módulo *Replayer* da ferramenta *WG-Modificada*, utilizando a ferramenta *Selenium WebDriver*;
   (b) Gerar relatório de cobertura, usando a ferramenta *Cobertura*;
   (c) Aplicar o *merge* entre os arquivos de cobertura; e
   (d) Selecionar o próximo dado de teste, ou seja, $w = w + 1$.

Tabela 5.3: *Conjunto de Dados de Teste por Aplicação – WG-Modificada X GAWG.*

Aplicações	*WG-Modificada (T)*	*GAWG (W)*	*WG-Modificada (t_{max})*	*GAWG (w_{max})*
Calendar	4	5	3	6
Carts	5	12	4	4
Checkbox	4	8	3	5
Colors	3	11	3	4

Segundo os dados apresentados da Tabela 5.3, observa-se que, para todas as aplicações, os conjuntos de dados de teste gerados pelo *GAWG* (W) têm mais dados de teste que o conjunto gerado pela própria *WG-Modificada* (T), ou seja, são maiores. O mesmo acontece com o tamanho do maior dados de teste para determinada aplicação, com exceção da aplicação *Carts*, que obteve o mesmo tamanho entre t_{max} e w_{max}. A princípio, isso pode ser visto como um ponto negativo para o *GAWG*, uma vez que se deseja um conjunto de dados de teste com uma menor quantidade de dados de teste e de preferência

que sejam menores possíveis. Na Seção 5.3.2.4 foram coletados dados de cobertura para os conjuntos de dados de teste das duas propostas para análise.

5.3.2.4 Análise dos Dados

Terminada a fase anterior, as informações relevantes para se atingir o objetivo proposto são coletadas a partir da base de dados construída durante a aplicação do estudo exploratório. Com isso, foram coletadas diferentes informações de modo a permitir a determinação de:

- Qual a cobertura dos dados de teste gerada pela *WG-Modificada*; e
- Qual a cobertura dos dados de teste gerada pelo *GAWG*.

Como exemplo, a Figura 5.11 ilustra o relatório de cobertura gerado para o programa *Calendar* utilizando ferramenta *Cobertura*. Esse relatório mostra algumas informações importante da SUT, como número de classes, quantidade e porcentagem de cobertura de LOC e de desvios e a complexidade, que é calculada com base na complexidade ciclomática definida na Seção 2.4.2.

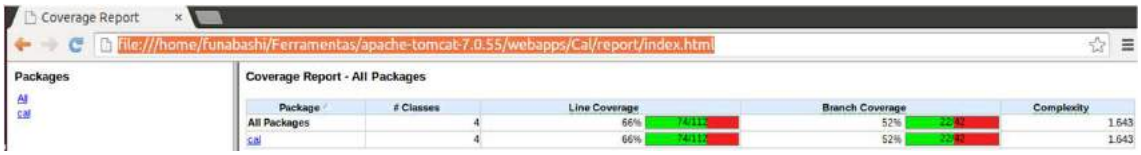


Figura 5.11: *Relatório Gerado pela Ferramenta Cobertura.*

A Tabela 5.4 apresenta os dados coletados com a aplicação do experimento. Observa-se que para cada aplicação foi calculada a cobertura de LOC e de desvios, tanto referente ao conjunto de teste gerado pela *WG-Modificada* quanto para o conjunto de teste gerado pelo *GAWG*. Um ponto que se destaca é que para a aplicação *Calendar* foram obtidas as mesmas coberturas, tanto pela *WG-Modificada* quanto pelo *GAWG*. Isso se justifica, pois essa aplicação faz o controle de autenticação, comum em aplicações web, esbarrando assim na limitação do módulo *Ripper* da ferramenta, citada na Seção 4.4.2, que não aplica o controle de verificação de dependência entre objetos, gerando assim um EFG incompleto da SUT.

Tabela 5.4: *Dados de Cobertura: WG-Modificada X GAWG.*

Aplicações	*WG-Modificada* *Cobertura de LOC*	*GAWG* *Cobertura de LOC*	*WG-Modificada* *Cobertura de Desvios*	*GAWG* *Cobertura de Desvios*
Calendar	66% (74/112)	66% (74/112)	52% (22/42)	52% (22/42)
Carts	60% (41/68)	72% (52/68)	42% (8/19)	63% (12/19)
Checkbox	69% (51/74)	88% (65/74)	56% (10/18)	78% (14/18)
Colors	47% (30/65)	91% (59/65)	40% (4/10)	80% (8/10)

Apesar dessa particularidade com a aplicação *Calendar* pelo problema encontrado na ferramenta com relação ao processo de *login*, em geral o *GAWG* se mostrou mais efetivo que a *WG-Modificada*. Observa-se que para todas as aplicações houve uma maior cobertura por parte do *GAWG*. Obteve-se uma média de 25% a mais de cobertura, levando em consideração todas as aplicações, sendo que para a aplicação *Colors* se tem um ganho na porcentagem de 44%. A cobertura de desvios segue o mesmo caminho, sendo uma média de 24,3%, aproximadamente, a mais de cobertura, com destaque para a aplicação *Colors* com um ganho de 40%. Apesar da soma de todos os dados de teste dos conjuntos gerados pela *GAWG* ser de, aproximadamente, duas vezes maior que a soma dos conjuntos gerados pela *WG-Modificada*, houve um ganho significativo de cobertura de LOC.

5.4 Considerações Finais

Neste capítulo foram apresentados conceitos referentes ao funcionamento dos Algoritmos Genéticos. Com o estudo desses conceitos, foi definido e implementado o *GAWG*, uma nova proposta de representação genética para customizar a geração dos dados de teste. Esta implementação foi feita dentro do fluxo de execução da *WG-Original*, visando evoluir o conjunto de dados de teste que já era gerado pela própria ferramenta, ou seja, substituir o gerador de dados de teste original da ferramenta *WG-Original* com o objetivo de se conseguir dados de teste com maior cobertura de código.

A ideia original desta tese era de evoluir a *WG-Original* por meio da inclusão de algoritmos de otimização visando à geração automática de dados de teste. Entretanto, durante os estudos exploratórios, observaram-se algumas limitações na ferramenta utilizada e principalmente no modelo que se utiliza para representar as aplicações em teste. Essas limitações apresentadas no módulo *Ripper* da ferramenta e no modelo utilizado motivaram a investigação de modelos mais abrangentes para representar WUIs. Dentro desse contexto, no Capítulo 6 é apresentada uma proposta de modelo que visa a superar algumas dessas limitações, proporcionando uma evolução do estado da arte no teste via UI, considerando os modelos de representação presentes atualmente.

CAPÍTULO 6

WUITAM – Modelo WUI para Automação dos Testes

6.1 Considerações Iniciais

Um modelo, dentro do contexto desse trabalho, é uma descrição gráfica ou textual do comportamento do sistema, fornecendo subsídios para prevê-lo e compreendê-lo. Recentemente, tem se pesquisado sobre teste e extração do modelo por meio de aplicações UI. Infelizmente, a maioria dessas pesquisas tem limitações e restrições sobre as aplicações UI que podem ser modeladas, prejudicando assim sua adoção pelo meio industrial. Memon et al. (2013) publicaram extensivamente sua pesquisa sobre GUI *Ripping*, uma técnica para extrair dinamicamente modelos baseados em eventos de aplicações GUI para fins de automação dos testes. Essa técnica é usada pela ferramenta de teste *Web GUITAR*, citada na Seção 4.4.2, e seu objetivo é fornecer meios para o processo de extração de modelo, proporcionando assim a geração de dados de teste automatizada. Porém, em suas pesquisas não são abordados problemas e desafios, como fornecer entradas específicas para determinados objetos; o grande número de estados para se modelar mesmo sistemas simples; e alternativas para redução desse número de estados (MEMON, 2007).

A mais recente abordagem para extração de modelo de UI é baseada em engenharia reversa dinâmica, ou seja, é feita a execução do aplicativo e se observa o comportamento em tempo de execução da UI. O grande desafio é passar automaticamente pela UI, fornecendo dados significativos para os campos de entrada requisitados, como usuário e senha válidos para uma tela de *login* sem instruções predefinidas do testador (intervenção humana) (AHO; MENZ; RATY, 2011). Geralmente, alguma intervenção humana é necessária durante o processo de modelagem para alcançar uma boa cobertura com modelos de engenharia reversa dinâmica (AHO; MENZ; RATY, 2011), o que significa que a modelagem é assistida manualmente por uma pessoa durante o processo de engenharia reversa ou o modelo inicial gerado é revisado, corrigido e estendido manualmente por uma pessoa após a extração do modelo de forma automática (KULL, 2012). Embora tenham sido pro-

postos processos semi-automáticos, em algumas situação não se é capaz de atingir todas as partes e/ou propriedades da UI durante a extração modelo (MEMON, 2007). De encontro com esse e outros problemas citados durante o texto desta tese, este trabalho define um meta-modelo para representação de WUIs, procurando dar subsídios para automação da atividade de teste de software, chamado *WUITAM* (***WUI Test Atomation Model***), que é detalhado neste capítulo.

6.2 Construção e Características do Meta-Modelo

O meta-modelo *WUITAM*, proposto neste capítulo, tem como objetivo aperfeiçoar as abordagens existentes na literatura referente à construção de modelos que representem, de uma forma mais completa, as características de interfaces gráficas no contexto web, permitindo assim a automatização, geração e execução de dados de teste. Uma das críticas ao MBT é com relação ao esforço necessário para construção dos modelos, a partir dos quais se geram os dados de testes. Essa complexidade aumenta se pensar no contexto de WUIs, pois se trata de uma tarefa demorada e muitas vezes não efetiva. Assim, torna-se importante identificar meios de automatizar o processo de teste de forma que as características de uma aplicação web sejam fielmente representadas. O meta-modelo proposto permite:

1. Gerar o modelo representando o maior número de características das aplicações web atuais;
2. Gerar os caminho de teste a partir dos modelos gerados; e
3. Futuramente, verificar a integridade do modelo construído.

Complementando as definições do Capítulo 3, as WUIs usam metáforas de objetos reais, como, botões, menus, *links*, guias, controles e ícones, para permitir a interação entre os usuários e a aplicação. Em relação ao nível do código, uma WUI é uma 3-tupla: $\{W,P,V\}$, na qual W é um conjunto desses objetos (*widgets*); P um conjunto de propriedades para cada objeto (tamanho, cor, fundo-cor, forma, por exemplo); e V um conjunto de valores válidos associados a cada propriedade de P. Para se alterar essas propriedades, são invocadas "operações", ou seja, eventos como cliques ou arrastar do mouse, preenchimentos via teclado, entre outros. Esses eventos são determinísticos e podem alterar o estado da WUI. Tecnicamente, essa representação de 3-tupla abrange um conceito conhecido como "estado WUI", que é um conjunto de valores para W, P e V, representando uma instância de uma WUI, em determinado instante de tempo.

Perante essas definições, esses conceitos foram estruturados no meta-modelo, relacionando-os em um diagrama que é apresentado na Figura 6.1. Foram determinados estados, os quais podem ser dos tipos: "*Start*", "*End*", "*Behavioural*" e "*Informative*".

O estado do tipo "*Start*" indica que a interação com a aplicação pode começar naquele instante, sendo que o estado "*End*" indica que aquele estado é considerado um estado final da aplicação. Os estados considerados "*Behavioural*" compreendem, como o próprio nome indica, estados que exibem algum tipo de comportamento e são os mais comuns em uma aplicação. Já o estado "*Informative*" indica que naquele momento alguma mensagem está sendo fornecida pelo sistema, como, por exemplo, um aviso de que um produto não está cadastrado.

Esses estados são formados por objetos, que podem ser obrigatórios ou não. Essa obrigatoriedade de uso influencia diretamente na geração de dados de teste executáveis, uma vez que se o objeto não for preenchido ou selecionado, a sequência requerida não dá continuidade.

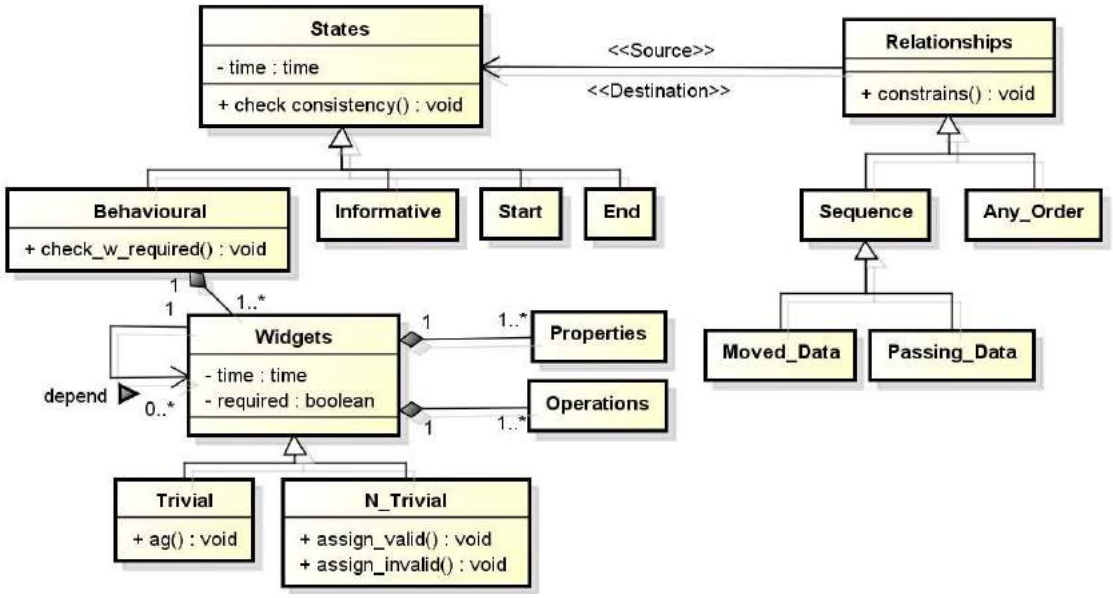


Figura 6.1: *WUITAM– Meta-Modelo para Representação de Aplicações Web.*

Pensando nas limitações das ferramentas estudadas e utilizadas neste trabalho, o meta-modelo dá subsídios para que sejam determinados quais objetos podem ser classificados como " *Trival*" e "*N_trivial*". Os objetos do tipo "*Trival*" são considerados objetos que não possuem dependência com outros objetos e que são dependentes apenas do tipo de dado nativo da linguagem de programação utilizada. Devido à essa característica, propõe-se o uso de um AG para geração automatizada dessas entradas, não simplesmente de forma aleatória, mas levando em consideração critérios existentes de teste. Porém, para que o AG seja proposto, para cada dos objetos devem ser definidas restrições. Por exemplo, para um campo de texto onde deve ser inserido um nome, define-se que são aceitos somente caracteres alfabéticos e que o campo deve ser preenchido obrigatoriamente (*required*). Já para uma caixa de seleção onde deve ser informada uma ou várias cores de

preferência, define-se que o campo é opcional (não *required*) e que aceita seleção múltipla. Tendo definidas as regras individuais, o próximo passo é identificar as dependências (*depend*), quando existentes, entre os objetos. Por exemplo, quando um determinado objeto só pode ser preenchido após a seleção de determinada opção. Muitas dessas restrições podem ser impostas pela linguagem de programação adotada, ou devido a detalhes de implementação, para respeitar as regras de negócio estabelecidas. Se implementado no AG a proposta do critério de particionamento em classes de equivalência, por exemplo, caso um objeto aceite valores numéricos até o número 20, então uma classe de equivalência seria de valores menores ou iguais a 20 (classe válida) e outra seria de valores maiores que 20 (classe inválida). Os dados de teste necessários para esse campo seriam um valor para cada classe de equivalência, como 1 e 25. Já os objetos do tipo "*N_Trivial*" aumentam, em muito, o custo da automatização do processo de geração dos dados de teste. Esses objetos possuem uma dependência não explícita em linguagens de programação, o que torna necessária a interação humana, aumentando assim o custo de todo o processo. Um exemplo são dois objetos x e y que representam, respetivamente, o $login$ e a *senha* de acesso a uma determinada aplicação web. A dependência entre x e y não é apenas de preenchimento por determinado tipo de dado ou limitação de quantidade de caracteres. A dependência seria entre um x relacionado a determinado y dentro de um banco de dados, difícil de ser identificada de forma automática.

Uma característica importante do modelo é que se pode ter tanto estados quanto objetos com temporização (*time*). Esse recurso é muito comum em aplicações web atuais, pois permite, durante um período determinado, que seja identificada sua inatividade ou expirada sua sessão. Exemplos reais de aplicação dessa temporização são os sites de bancos (internet *bankings*), que a aplicam procurando uma maior segurança.

No que se refere aos relacionamentos aplicados aos possíveis estados, intitula-se como "*Relationships*", foram determinados 3 tipos, sendo que o tipo "*Sequence*", que se refere a uma ordem nos acontecimentos, possui dois subtipos, "*Passing data*" e "*Moved Data*". Na relação "*Passing Data*", existe passagem de informação do elemento de origem para o elemento de destino, mantendo a possibilidade de uso da informação pelo primeiro. Já a relação "*Moved data*" modela uma situação em que o elemento de origem passa informação para o elemento de destino, perdendo o primeiro essa mesma informação. Finalmente, a relação "*Any Order*" indica que os elementos de destino podem ser executados em qualquer ordem. Essas formas de relacionamento foram adaptadas do trabalho de Moreira e Paiva (2014b) que, para construção da ferramenta proposta em seu trabalho, determinaram os principais tipos de relacionamento entre estados de navegação em uma aplicação web.

Algumas regras impostas para a aplicação do meta-modelo:

1. Só pode existir um nó do tipo *Start* e vários nós do tipo *End* por modelo;

2. Nós (estados) do tipo *Start* não podem ser destino de uma relação;
3. Nós do tipo *End* não podem ser origem de uma relação;
4. Não podem existir nós sem nome ou número identificativo;
5. Quando existe passagem de informação entre nós, as configurações presentes na relação de passagem de informação devem ser coerentes com as configurações existentes nos nós; e
6. Todo objeto não trivial (*N_Trivial*) deve ter, pelo menos, um conjunto de valores válido e um inválido fornecido pelo testador. Considerando o exemplo do *login* e *senha* , tais conjuntos poderiam ser: $login = joao$ e $senha = klsbt$, como um conjunto inválido e, $login = joao$ e $senha = AdminTec$, como um conjunto válido.

Cada página web é implementada e corresponde a determinadas funções da aplicação da web. Páginas web são relacionadas pela maneira que são acessadas entre si, ou seja, pela maneira que uma página "chama" outra página, tais como: ações de clique em *links*, botões ou guias. Baseando nessa definição e no meta-modelo proposto neste trabalho, os comportamentos de UI de cada página da web são representados por uma máquina de estado estendida definida como: $M = < S, s_0, \Sigma, \delta, S_n >$, onde:

- S é um conjunto finito não vazio de estados para uma página web;
- $s_0 \in S$ é o estado inicial;
- Σ uma coleção de eventos [condição];
- δ uma função de transição de estado, onde $\delta : S \times \Sigma \rightarrow S$; e
- $F \subseteq S$ é um conjunto de estados finais.

Para a construção da máquina de estado estendida, foram formalizadas, por meio de "simbolos" reservados, as regras impostas pelo meta-modelo da Figura 6.1. Esses símbolos e suas descrições são apresentadas na Tabela 6.1:

Tabela 6.1: *Simbologia para Construção da Máquina de Estados Estendida.*

	Símbolos	**Descrição**
Objetos (*Widgets*)	**&**	Objeto Obrigatório (*Required*)
	*	Não Trivial (*N_Trivial*)
	N	Dependência entre Objetos
Estados (*States*)	**#**	Comportamental (*Behavioral*)
	$	Informativo (*Informative*)
Relacionamentos (*Relationships*)	@	Passar de Dados (*Passsing Data*)
	|	Mover Dados (*Moved Data*)
	%	Qualquer Ordem (*Any Order*)

Para geração dos dados de teste, é adotada a sintaxe $s_0 * event_i = S_i * ... = S_{final}$. Ele é iniciado pelo estado inicial s_0, sendo que neste estado, se o $event_i$ ocorrer, a página tem seu estado alterado para o s_i. Quando se atinge um estado final, tem-se um dado de teste. O algoritmo proposto para gerar dados de teste satisfaz as seguintes exigências:

- Todos os estados e transições do modelo devem ser cobertos; e
- O resultado é uma lista de caminhos de teste distintos, que começam no estado inicial e terminam em um estado final.

6.3 Estudo Exploratório

Nesta Seção será apresentado o estudo exploratório cujos objetivos são verificar se o meta-modelo apresentado fornece as características necessárias para representação da interface de uma aplicação web e, com isso, gerar dados de teste satisfatórios. O processo desenvolvido consiste em utilizar a máquina de estados estendida, gerada, baseando-se no meta-modelo apresentado na Figura 6.1, com o objetivo de gerar possível diferentes caminhos, que neste contexto representam ações a serem executadas na aplicação web.

Para verificar a conquista dos objetivos descritos, foram determinadas duas questões de pesquisa:

QP_1: O meta-modelo tem as características necessárias para representar uma aplicação web?

QP_2: O meta- modelo fornece subsídios para geração de dados de teste?

6.3.1 Aplicação do *WUITAM*

Buscando responder às QP_1 e QP_2 apresentadas, foi utilizado, para aplicação do *WUITAM*, como exemplo, o controle de acesso (*login*) do serviço de *webmail*[1] da Universidade Federal de Mato grosso do Sul (UFMS). Como mostra a Figura 6.2, a aplicação possui apenas três objetos, sendo duas caixas de texto e um botão. Porém, a aplicação apresenta especificações que dificultam a sua representatividade por meio de um modelo simples. São elas:

- O preenchimento do "*Login*" e da "*Senha*" podem ser feitos em qualquer sequência;
- Caso seja acionado o botão "*Entrar*", sem o "*Login*" e a "*Senha*" preenchidos, a aplicação deve indicar seus preenchimentos na sequência "*Login*" e "*Senha*";
- Caso seja acionado o botão "*Entrar*", sem o campo "*Senha*" preenchido, a aplicação deve indicar o preenchimento;
- Caso seja acionado o botão "*Entrar*", sem o campo "*Login*" preenchido, a aplicação deve indicar o preenchimento;
- Caso seja acionado o botão "*Entrar*", com o "*Login*" ou a "*Senha*" preenchidos com valores inválidos, a aplicação deve informar o erro e solicitar um *Login* e *Senha* válidos; e

[1] https://webmail.ufms.br/

- Caso seja acionado o botão "*Entrar*", com o "*Login*" e a "*Senha*" preenchidos com valores válidos, o controle de acesso deve ser finalizado.

Figura 6.2: *Webmail da UFMS.*

Procurando responder à QP_1, uma máquina de estados estendida – utilizando o meta-modelo, para sua construção, da página em questão – é gerada e apresentada na Figura 6.3. Observa-se que o relacionamento que parte do estado inicial sugere que a manipulação dos objetos, no estado $s1$, pode ser aplicada em qualquer ordem (#). Na tela inicial, representada pela estado $s1$, existem 3 possibilidade de eventos:

1. Caso o Usuário (*t_usuario*) seja preenchido ($t\_usuario.filled$) e qualquer tecla de próximo seja pressionada (*next*) ou o botão Entrar (*b_entrar*) seja clicado (*b_entrar.set*), então se atinge o estado $s2$, indicando que os dados do estado $s1$ foram passados para $s2$ (@);
2. Caso o Senha (*t_senha*) seja preenchida ($t\_senha.filled$) e qualquer tecla de próximo seja pressionada (*next*) ou o botão Entrar (*b_entrar*) seja clicado (*b_entrar.set*), então se atinge o estado $s3$, indicando que os dados do estado $s1$ foram passados para $s3$ (@); e
3. Caso o Usuário (*t_usuario*) e a Senha (*t_senha*) não sejam preenchidos e o botão Entrar (*b_entrar*) seja clicado (*b_entrar.set*), então se permanece no estado $s1$.

A mesma interpretação se aplica aos outros estados e eventos. Observa-se que apenas o estado $s2$ é considerado, além de um estado comportamental (#), um estado informativo ($). Isso se deve ao fato da aplicação só informar a mensagem "Falha de *Login*" quando o *Login* ou a *Senha* é inválido, direcionando para o estado $s2$ que solicita uma nova *Senha*, independente se o *Login*, a *Senha* ou ambos que estão incorretos.

6.3.2 Resultados Experimentais

Para responder à QP_2, é proposto um algoritmo de busca em profundidade para geração de possíveis dados de teste (caminhos). Nesse algoritmo (Algorítmo 6.1)

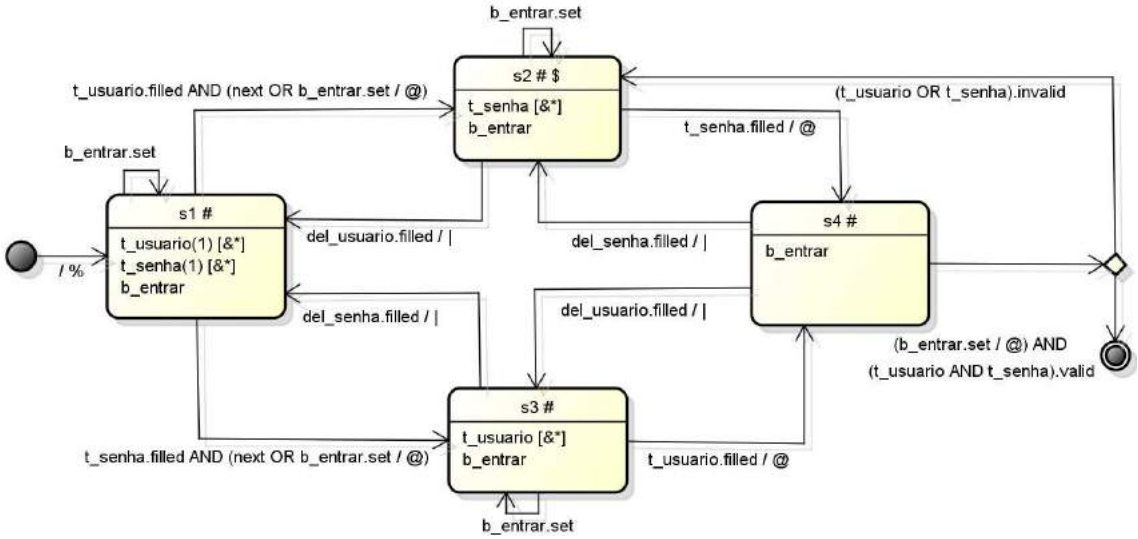


Figura 6.3: *Máquina de Estado Estendida – Webmail da UFMS.*

a variável de entrada é o estado inicial i e o caminho é armazenado na variável P. Inicialmente a variável *back* é inicializada com o valor $true$ (linha 4). Posteriormente, será verificada cada transição (linha 5). Se a transição não foi "aprovada" (linha 6) serão verificados *widgets* com dependências ou não triviais (linha 7), caso a verificação seja positiva, será necessária a intervenção do testador (linha 8). Neste caso, o testador deve fornecer ao menos uma entrada válida e uma inválida. Senão, caso a verificação seja negativa, os dados podem ser gerados utilizando uma meta-heurística, como um algoritmo genético, por exemplo (linha 10). O valor da variável *back* recebe $false$ (line 11). Em seguida, o algoritmo irá adicionar o estado inicial para aquela transição em uma lista $LIST$, sendo utilizada essa variável para armazenar de forma temporária os caminhos e registrar que transição foi "aprovada" (linha 13), ou seja, se a transição a partir daquele estado ainda não foi utilizada no caminho em criação, não aceitando mais de uma mesma transição que parte do mesmo estado no mesmo caminho. O algoritmo trabalha de forma recursiva com a entrada, sendo o estado final da transição "aprovada" (linha 14), até que não é possível adicionar mais transição nos caminhos de $LIST$. Finalmente, todos os estados de $LIST$ são removidos para se criar um novo P (linha 15) e se o valor da variável *back* é $true$ (linha 17), a variável $LIST$ é acrescentada à variável P (linha 18).

Com a aplicação do algoritmo, temos alguns caminhos que são apresentados na Tabela 6.2.

Mesmo para uma aplicação simples, observa-se que o número de possíveis caminhos pode ser grande. Assume-se que, para reduzir o número de caminhos gerados, se uma transição $s_i * event_j$ já ocorreu em determinado caminho, ela não poderá ocorrer novamente no caminho em questão. Focando ainda na redução do conjunto de dados de teste (caminhos), procurando manter a maior cobertura de estados e transições, foi analisada

Algoritmo 6.1 D_Search – Algoritmo para Geração de Caminhos.

```
1  Entrada i: estado inicial
2  Entrada P: conjunto de caminhos
3  Saída lista de caminhos de teste
4  boolean back = true;
5  para todos eventos faça
6    se a transição não foi aprovada então
7      se o estado tem objeto com dependência ou N_Trivial então
8        interação com o testador;
9      senão
10       meta-heurística( );
11       back = false;
12       a transição foi aprovada;
13       adicinado estado_event em LIST;
14       D_Search(j,P);
15       remover todos os estados de LIST;
16     fimse
17     se back = true então
18       adiciona LIST em P;
19     fimse
20   fimse
21 fimpara
```

Tabela 6.2: *Parte da Lista de Possíveis Caminhos.*

#	Path
1	s1*t_usuario.filled AND Next=s2*t_senha.filled=s4*b_entrar.set*(t_usuario AND t_senha).valid = end
2	s1*b_entrar=s1*t_usuario.filled AND Next = s2*t_senha.filled=s4*b_entrar.set*(t_usuario AND t_senha).valid = end
3	s1*b_entrar=s1*t_usuario.filled AND Next = s2*del_usuario.filled=s1*b_entrar.set*(t_usuario AND t_senha).valid = end
4	s1*b_entrar=s1*t_usuario.filled AND b_entrar.set = s2*del_usuario.filled=s1*b_entrar.set*(t_usuario AND t_senha).valid = end
5	s1*b_entrar=s1*t_usuario.filled AND b_entrar.set=s2*t_senha.filled=s4*b_entrar.set*(t_usuario AND t_senha).invalid=s2*b_entrar.set=s2*del_usuario.filled=s1*t_senha.filled AND b_entrar.set=s3*b_entrar.set=s3*t_usuario.filled=s4*b_entrar.set*(t_usuario AND t_senha).valid = end
6	s1*b_entrar=s1*t_usuario.filled AND Next = s2*t_senha.filled=s4*b_entrar.set*(t_usuario AND t_senha).valid = end
7	s1*t_senha.filled AND b_entrar.set=s3*t_usuario.filled=s4*b_entrar.set*(t_usuario AND t_senha).valid=end
8	s1*b_entrar=s1*t_senha.filled AND b_entrar.set=s3*t_usuario.filled=s4*b_entrar.set*(t_usuario AND t_senha).invalid=s2*b_entrar.set=s2*del_usuario.filled=s1*t_usuario.filled AND b_entrar.set=s2*b_entrar.set=s2*t_senha.filled=s4*b_entrar.set*(t_usuario AND t_senha).valid = end
n	.....

a abordagem da árvore de alcançabilidade, proposta por Masiero, Maldonado e Boaventura (1994). A árvore de alcançabilidade consiste em gerar o comportamento possível do sistema modelado, ou seja, é formada pelo conjunto de estados (ou configurações) possíveis do sistema. A partir do estado inicial do sistema, todas as transições disparáveis são representadas juntas aos estados alcançados. Para cada novo estado inserido na árvore, são obtidas as transições disparáveis e os estados alcançados e assim sucessivamente, até

que todos os estados alcançáveis, a partir do estado inicial, sejam representados. É importante ressaltar que o conjunto de alcançabilidade pode ser finito, no entanto, é comum encontrarmos situações nas quais as marcações alcançadas não nos levam a um estado inicial, ou à marcação inicial, a raiz de nossa árvore.

Ainda segundo Masiero, Maldonado e Boaventura (1994), a utilização de árvore de alcançabilidade possibilita que algumas propriedades dinâmicas do sistema sejam investigadas, tais como:

- **Validade de uma sequência de eventos:** uma sequência de eventos é válida se cada evento leva ao disparo de uma transição, produzindo uma mudança de configuração do sistema;
- **Alcançabilidade de estados globais:** um estado global S_k é alcançável a partir de um estado global S_i se existe no mínimo uma sequência de eventos válida seq_1 e uma possível sequência de estados intermediários tal que S_k possa ser obtido a partir de S_i, percorrendo a sequência seq_1;
- **Reiniciabilidade:** um modelo é reiniciável quando para cada estado global S_i existe uma sequência de eventos válida que levam ao estado inicial S_0; e
- **Uso de transição:** uma transição é usada se ela aparece em, no mínimo, um caminho da árvore de alcançabilidade.

Este método envolve essencialmente a enumeração de todas as marcações alcançáveis feitas por meio de um grafo do tipo árvore em que os nós são do tipo vetores de estado, alcançados sucessivamente e alternativamente pela rede, e os arcos são as correspondentes transições executadas. Entretanto, sua aplicabilidade é comprometida devido ao problema de explosão de estados, pois, mesmo considerando sistemas pequenos, o número de estados possível pode ser muito grande, gerando um alto custo para a construção da árvore. Procurando minimizar este problema, o trabalho de Barnard (1998) cita algumas técnicas de redução que podem ser aplicadas durante a construção da árvore. Duas dessas técnicas que se destacam são:

- **Nós duplicados:** utilizada para representar estados repetidos na árvore. Quando um estado já existente é inserido, ele é considerado apenas um *link* para a primeira ocorrência desse estado, não sendo gerados novamente seus estados sucessores; e
- **Conjunto de componentes:** considera apenas alguns componentes do sistema modelado durante a construção da árvore de alcançabilidade.

Aplicando o conceito de nós duplicados para redução da árvores, a Tabela 6.3 e a Figura 6.4 mostram, respetivamente, a lista de eventos (transições) e a árvore de alcançabilidade para a máquina estendida da Figura 6.3.

Tabela 6.3: *Lista de Eventos (Transições).*

Identificação	Evento
a	$b\_entrar.set$
b	$t\_usuario.filled\ AND\ (next\ OR\ b\_entrar.set/@)$
c	$t\_senha.filled\ AND\ (next\ OR\ b\_entrar.set/@)$
d	$del\_usuario.filled/\|\|$
e	$del\_senha.filled/\|\|$
f	$t\_usuario.filled/@$
g	$t\_senha.filled/@$
h	$(t\_usuario\ OR\ t\_senha).invalid$
i	$(b\_entrar.set/@)\ AND\ (t\_usuario\ AND\ t\_senha).valid$

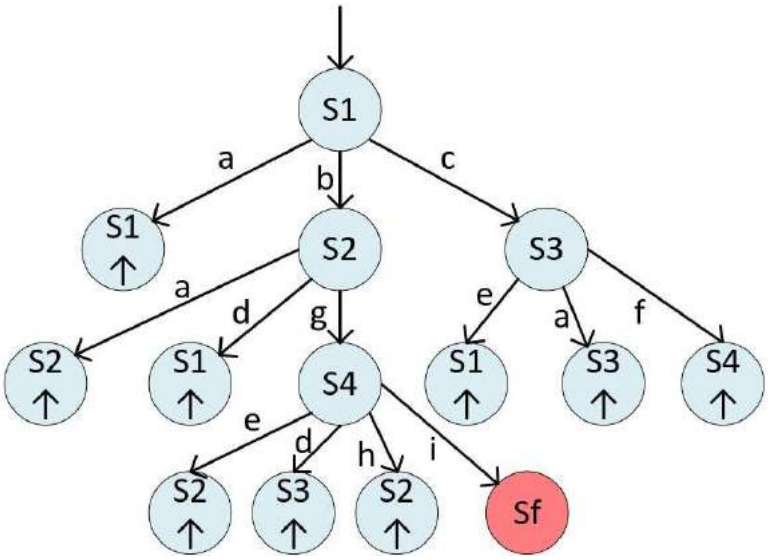


Figura 6.4: *Árvore de Alcançabilidade de Cardinalidade 1.*

Para a árvore de alcançabilidade da Figura 6.4, foi assumida a cardinalidade 1, ou seja, caso um estado já ocorreu uma vez na construção da árvore, ele já é colocado como "interrompido", ou seja, a partir dele, não será dado prosseguimento na construção da árvore. Esse estado é representado pelo símbolo ↑. Com essa cardinalidade, se observa a possibilidade de gerar muitos caminhos "parciais", ou seja, caminhos que não atingem o estado Sf . Apenas o caminho $S1 \rightarrow S2 \rightarrow S4 \rightarrow Sf$ seria capaz de cobri-lo. Esses caminhos parciais podem ser determinados como critério pelo testado, por exemplo, gerar caminhos que alcancem os estados $S3$ da árvore.

Pensando em uma menor restrição na geração desses caminhos, a cardinalidade foi aumentada para 2. Com isso, para um estado ser determinado como parcial, sua ocorrência na árvore deve ter acontecido duas vezes, para ai sim, na terceira, ser denominado como estado parcial. Utilizando essa cardinalidade, foi gerada a árvore da Figura 6.5. Fica claro que, por meio dessa árvore, é possível a geração de mais caminhos com determinado objetivo de cobertura, sendo possível reduzir e aumentar, de forma controlada, o tamanho do conjunto de dados de teste.

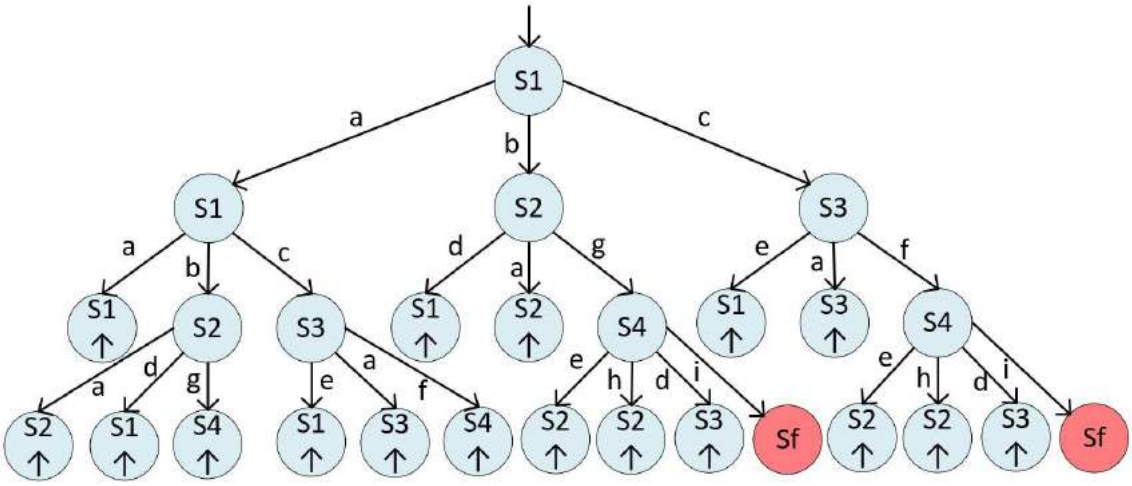


Figura 6.5: *Árvore de Alcançabilidade de Cardinalidade 2.*

Outra abordagem que se pode usar é a Abordagem de Similaridade (*Weighted Similarity Approach* – WSA), proposta por Bertolino et al. (2008). Essa abordagem utiliza funções de similaridade e pesos para a seleção de dados de teste. Segundo Bertolino et al. (2008), ao observar um conjunto de dados de testes gerado, é normal a identificação de um número considerável de dados de teste redundantes, ou seja, cobrem o mesmo conjunto de funcionalidades (estados e transições). Estes dados de teste semelhantes podem ser excluídos do conjunto de teste, sem comprometer, por exemplo, a cobertura das funcionalidades. Assumem-se os dados de teste mais importantes aqueles que mais se diferenciam dos outros. Os que possuem uma certa similiraridade devem ser analisados em pares, sendo um descartado por pontuação. Por exemplo, o $path_7(S_I * EVENT_J) \subseteq path_5(S_I * EVENT_J)$, portanto $path_5$ pontua em 1, enquanto $path_7$ não recebe pontuação. Definindo, para cada $path_W(S_I * EVENT_J) \subseteq path_Z(S_I * EVENT_J)$, $path_Z$ é pontuado com 1 para futura sequência de execução dos dados de teste. Escolhe-se os mais pontuados para formação do conjunto de dados de teste.

Na próxima seção, foi feita uma análise para comparação do *WUITAM* com as duas principais ferramentas estudadas e encontradas na literatura para geração de dados de teste a partir de seus modelos nativos, a *WG-Modificada* e a *PBGT*.

6.3.3 Análise Final das Ferramentas *WG-Modificada* X *PBGT* X *WUITAM*

A geração de dados de teste a partir de modelos é considerada um desafio, uma vez que representa um impacto considerável sobre o custo da atividade de teste de software. Devido a não verificação e controle de dependência entre objetos, a ferramenta *WG-Modificada* aplicou seu módulo *Ripper* na aplicação de *webmail* da UFMS, obtendo

um EFG incompleto, uma vez que não conseguiu determinar entradas de dados válidas para criação completa do grafo. Com isso, foram gerados apenas quatro dados de teste: *Login* em branco e *Senha* em branco; *Login* com um texto qualquer e *Senha* em branco; *Login* em branco e *Senha* qualquer; e os dois campos preenchidos, mas sem se preocupar com a dependência desses dados.

No contexto da ferramenta *PBGT*, dados de teste são gerados automaticamente a partir de um modelo construído na ferramenta *PARADIGM*. Um modelo pode ser visto na forma de um grafo, com ilustrado na Figura 6.6, que representa a aplicação de *webmail* da UFMS. Esse modelo é constituído por um conjunto de nós, chamados de elementos (*elements*), e ligadas por setas, chamadas de conectores (*connectors*).

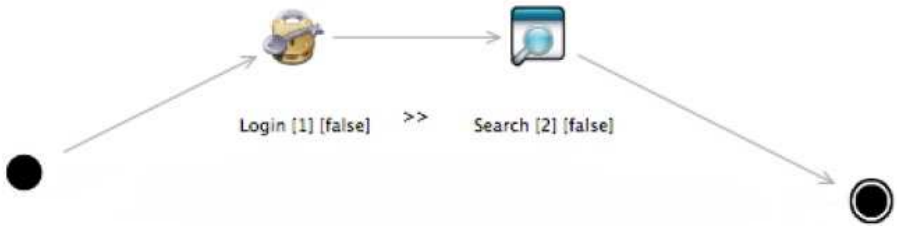


Figura 6.6: *Modelo Gerado pela Ferramenta PARADIGM – Webmail da UFMS.*

O primeiro passo consiste em extrair todos os padrões de teste de interface do usuário. Em seguida, o componente *PARADIGM–TG* irá gerar todos os caminhos possíveis que atravessam o modelo, a partir do estado inicial para o estado final, ou seja, um caminho representa uma sequência de padrões de teste de UI (NABUCO; PAIVA, 2014). Porém, como os dados de *Login* e *Senha*' são fortemente dependentes, é necessário que durante a configuração o testador tenha que fornecer dados de entrada como o login e a senha válidos para "*Valid Login*" e login e senha inválidos para "*Invalid Login*". Isso é executado como mostrado na Figura 6.7.

Dentro desse contexto, existem algumas características inseridas nos modelos gerados pela ferramenta *PARADIGM* que devem ser discutidas. Um ponto é que a ferramenta não fornece subsídios para identificar se *Login*, ou se a *Senha*, ou ambos são inválidos, prejudicando assim a geração de dados de teste específicos e que, como consequência, em algumas situações não cobrirão determinadas partes do código. Além disso, apesar da aplicação em teste não aplicar o conceito de temporização (*temporary*), a ferramenta *PGBT* não dá subsídios para representação dessa característica.

No termino da execução, foram gerados dois dados de teste, sendo um dado de teste válido e outro inválido, não se preocupando com alguns estados que a aplicação pode assumir em determinados momentos, como por exemplo, um *Login* preenchido e a *Senha* deixada em branco.

Figura 6.7: *Entrada de Valores Válidos e Inválidos – PBGT.*

A ferramenta *PBGT* se encontra em um estado avançado de desenvolvimento, mas algumas características de aplicações web atuais devem ser ainda implementadas. Observa-se ainda uma falta de suporte para "padrões", isto é, existem elementos em interfaces de aplicações web que não suportam os *patterns* já definidos e, com isso, irão prejudicar a geração e execução de determinadas ações.

6.4 Considerações Finais

Com o uso crescente da web, a necessidade de produção de aplicações com alta qualidade tornou-se mais evidente a busca por novos recursos, consequentemente, aumentou sua complexidade. Aplicações web têm sido um desafios para as pesquisas na área de Engenharia de Software e de Teste de Software em particular. As tecnologias envolvidas evoluíram rapidamente, com o advento das arquiteturas orientadas a serviços, computação baseada em nuvem e software auto-adaptativo, trazendo assim desafios para a atividade de teste software. Este capítulo apresentou detalhes do modelo *WUITAM*, fornecendo subsídios para automatização dos testes de aplicações web. As conclusões finais, contribuições desse trabalho e propostas de trabalhos futuros são apresentadas no próximo capítulo.

CAPÍTULO 7

Conclusões

Neste capítulo é feita uma análise global do trabalho realizado, pontuando as principais contribuições e propostas de trabalhos futuros. Esta tese, inicialmente tinha como objetivo principal propor uma técnica para geração de dados de teste por meio da interface gráfica do usuário de uma aplicação com foco não em testar sua usabilidade, mas sim verificar se os dados de teste gerados a partir dessas interfaces possuem uma relação com cobertura de código-fonte, funcionalidades do sistema ou qualquer outra premissa proposta pelos critérios de teste já existente. Com isso, percebeu-se a necessidade de uma revisão da literatura para verificar se existiam pesquisas com o mesmo foco proposto. Procurando aplicar uma revisão da literatura sistematizada, foi desenvolvido um mapeamento sistemático do qual se concluiu que a maioria dos trabalhos tinham foco no teste da interface gráfica do usuário, porém no sentido de usabilidade, sendo que poucos trabalhos estavam preocupados com as relações inicialmente propostas para investigação nesta tese.

Perante os fatos identificados nos estudos do mapeamento sistemático, foi percebida a necessidade de desenvolvimento de um algoritmo genético que fosse capaz de gerar dados de teste a partir de uma UI, mas que funcionasse em conjunto com ferramentas já implementadas, no caso a *Web GUITAR*, e que fosse capaz de fornecer métricas para se medir cobertura de LOC, por exemplo. Dentro desse contexto, foi implementado o *GAWG*, que se trata de um AG adaptado para o fluxo de execução da ferramenta *WG-Modificada* e que foi utilizado como substituto do gerador de dados de teste original da ferramenta. Com a aplicação do estudo exploratório, verificou-se que, para três das quatro aplicações utilizadas no estudo, houve uma maior cobertura de LOC por parte do *GAWG*, obtendo-se uma média de 25% a mais de cobertura, levando em consideração todas as quatro aplicações. A particularidade de uma das aplicações fez com que algumas limitações na *WG-Modificada* fossem observadas, com relação a sua utilização e, principalmente, com o modelo escolhido pela ferramenta para representar as aplicações em teste. Essas limitações apresentadas motivaram o estudo e a definição de um modelo de representação da interface gráfica que pudesse ser gerado de forma automática ou semi-automática e servisse como base para a geração de dados de teste. Dentro desse contexto,

foi proposto o *WUITAM*, procurando aproveitar as vantagens de teste baseado em modelos e, simultaneamente, "atacar" o problema da construção de modelos mais reais e completos. Comparado com as propostas encontradas na literatura, a máquina de estado gerada por meio do meta-modelo *WUITAM* representa, de forma mais fiel, os detalhes de uma aplicação web, como objetos com dependência de uso e dependências correlatas.

7.1 Contribuições

1. Uma grande contribuição deste trabalho, que foi apresentada no Capítulo 4, é fornecer uma melhor compreensão sobre as conceitos e aplicabilidade de teste de interface gráfica, por meio de um mapeamento sistemático que possibilita à comunidade científica novas estratégias de pesquisas sobre esse contexto, de acordo com a identificação das lacunas e descrição de alguns trabalhos;
2. Considerando a importância da automatização, mesmo que parcial, na geração de dados de teste, o algoritmo genético *GAWG*, proposto e descrito no Capítulo 5, fica como uma das maiores contribuições deste trabalho. Abre-se, assim, uma gama de opções de pesquisa;
3. Outra contribuição consiste no meta-modelo proposto, o *WUITAM*. O meta-modelo, quando aplicado na representação de interfaces gráficas, pode ser utilizado tanto para GUI quanto para WUI, pois fornece subsídios para representar características tanto de aplicações *desktop* quanto de aplicações web. No Capítulo 6, foi comprovado que, por meio do modelo, é possível gerar caminhos que percorram determinados estados e arestas, garantindo assim que, alguns comportamentos de uso da interface sejam testados. Porém, o conjunto desses caminhos pode ser muito grande, tornando, em algumas situações, necessária a aplicação de um processo de seleção. Dentro desse contexto, este trabalho também contribuiu com a proposta de uma adaptação no algoritmo, proposto por Masiero, Maldonado e Boaventura (1994), para geração da árvore de alcançabilidade e de sequências executáveis.

Por fim, o trabalho também contribui para os praticantes de desenvolvimento de software em geral e especificamente para os testadores de software, que poderão utilizar tanto o *GAWG* quanto o *WUITAM* para auxiliar na geração e determinação de dados de teste. Além disso, o mapeamento sistemático também pode ser utilizado como referencial bibliográfico para novas pesquisas nessa área.

7.2 Trabalhos Futuros

O protocolo utilizado para aplicação do mapeamento sistemático pode ser atualizado. Para isso, sugere-se que as *strings* de buscas sejam aperfeiçoadas e reaplicadas, incluindo mais palavras-chave e, com isso, atualizando e melhorando o levantamento das pesquisas relacionadas ao teste UI.

Como proposta geral de trabalhos futuros, pode-se citar a realização de mais estudos exploratórios envolvendo tanto o *GAWG* quanto o *WUITAM*. No caso do AG, esses estudos podem ser conduzidos medindo-se a evolução de cobertura de LOC de cada população gerada, variando os parâmetros de tamanho da população, taxa de cruzamento e a taxa de mutação. Além disso, alguns parâmetros podem ser modificados e implementados, como:

- Aplicar novas formas de representação de cromossomos;
- Mudar os parâmetros como a taxa de mutação e cruzamento, tamanho da população e critério de parada para futuras comparações;
- Implementar outros métodos de seleção como, por exemplo, *Ranking* e Torneio;
- Estudar outros operadores de cruzamento e mutação;
- Implementar outras funções-objeto com o intuito de verificar a cobertura de funcionalidades (teste caixa preta) e satisfazer critérios de teste baseados em defeitos, como o critério análise de mutantes; e
- Implementar outras meta-heurísticas, fornecendo assim subsídios para outros estudos exploratórios de comparação com o AG já implementado.

Percebe-se que, com a solução implementada no *GAWG*, muito ainda pode ser feito. Apesar da implementação ter sido estruturada utilizando os parâmetros da ferramenta *WG-Modificada*, pode ser adaptado ou implementado para outras ferramentas existentes no mercado, como a *PBGT* e *Murphy*. Com isso, pode-se realizar mais estudos exploratórios que possam ser conduzidos para se medir e comparar a evolução de cobertura de LOCs de cada população gerada em cada ferramenta.

Com relação ao *WUITAM*, além da possibilidade de ser utilizado em aplicações maiores, permitindo analisar sua eficácia com relação a geração de dados de teste, fica como sugestão de trabalhos futuros:

- Criar de uma ferramenta gráfica para a criação do modelo de forma a impedir o utilizador de cometer erros e de criar máquinas de estado inválidas;
- Melhorar o algoritmo de geração de caminhos junto ao modelo;
- Estudar formas de “atacar” o problema de explosão de estados; e
- Implementar e aplicar meta-heurísticas para geração e seleção de dados de teste junto ao *WUITAM*.

Referências Bibliográficas

AHO, P.; MENZ, N.; RATY, T. Enhancing generated java gui models with valid test data. In: *Open Systems (ICOS), 2011 IEEE Conference on*. [S.l.: s.n.], 2011. p. 310–315.

AHO, P. et al. Automated java gui modeling for model-based testing purposes. In: *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*. [S.l.: s.n.], 2011. p. 268–273.

AHO, P.; RATY, T.; MENZ, N. Dynamic reverse engineering of gui models for testing. In: *Control, Decision and Information Technologies (CoDIT), 2013 International Conference on*. [S.l.: s.n.], 2013. p. 441–447.

AHO, P. et al. Murphy tools: Utilizing extracted gui models for industrial software testing. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*. [S.l.: s.n.], 2014. p. 343–348.

AHO, P. et al. Industrial adoption of automatically extracted gui models for testing. In: *EESSMOD@MoDELS*. [S.l.]: CEUR-WS.org, 2013. (CEUR Workshop Proceedings, v. 1078), p. 49–54.

AHO, P. et al. Making gui testing practical: Bridging the gaps. In: *Information Technology - New Generations (ITNG), 2015 12th International Conference on*. [S.l.: s.n.], 2015. p. 439–444.

ALI, S. et al. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 36, n. 6, p. 742–762, nov. 2010. ISSN 0098-5589.

ALSHRAIDEH, M.; BOTTACI, L. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, John Wiley and Sons Ltd., Chichester, UK, v. 16, n. 3, p. 175–203, 2006. ISSN 0960-0833.

ALSMADI, I. *Building a Gui Test Automation Framework Using the Data Model*. [S.l.]: VDM Verlag, 2008.

ALSMADI, I. Using genetic algorithms for test case generation and selection optimization. In: *Electrical and Computer Engineering (CCECE), 2010, 23rd Canadian Conference on*. [S.l.: s.n.], 2010. p. 1–4.

ALSMADI, I.; MAGEL, K. Gui path oriented test generation algorithms. In: *Proceedings of the Second IASTED International Conference on Human Computer Interaction*. Anaheim, CA, USA: ACTA Press, 2007. (IASTED-HCI '07), p. 216–219. ISBN 978-0-88986-655-3.

AMALFITANO, D. et al. Exploiting the saturation effect in automatic random testing of android applications. In: *The Proceedings of the 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft 2015)*. [S.l.: s.n.], 2015.

AMALFITANO, D. et al. A toolset for gui testing of android applications. In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. [S.l.: s.n.], 2012. p. 650–653. ISSN 1063-6773.

AMALFITANO, D. et al. Using gui ripping for automated testing of android applications. In: *ASE '12: Proceedings of the 27th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2012.

AMALFITANO, D. et al. Mobiguitar – a tool for automated model-based testing of mobile apps. *IEEE Software*, IEEE Computer Society Press, Los Alamitos, CA, USA, 2014.

APACHE, S. F. *Apache Tomcat*. 2016. Página WEB. Disponível on-line: http://tomcat.apache.org/. Acesso em: 08-02-2016.

ARLT, S.; BERTOLINI, C.; SCHäF, M. Behind the scenes: An approach to incorporate context in gui test case generation. In: *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. Washington, DC, USA: IEEE Computer Society, 2011. (ICSTW '11), p. 222–231. ISBN 978-0-7695-4345-1.

AVASARALA, S. *Selenium WebDriver Practical Guide*. [S.l.]: Packt Publishing, 2014. ISBN 1782168850, 9781782168850.

BäCK, T. Optimal mutation rates in genetic search. In: *Proceedings of the 5th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. p. 2–8. ISBN 1-55860-299-2.

BALUJA, S.; CARUANA, R. *Removing the Genetics from the Standard Genetic Algorithm*. Pittsburgh, PA, USA, 1995.

BAOLA; NEURONADE. *Jabref – Open Source Bibliography Reference Manager*. 2016. Página WEB. Disponível on-line: http://www.jabref.org/. Acesso em: 24-02-2016.

BARESEL, A. et al. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In: *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2004. p. 108–118. ISBN 1-58113-820-2.

BARNARD, J. Comx: a design methodology using communicating x-machines. *Information and Software Technology*, v. 40, n. 5?6, p. 271 – 280, 1998. ISSN 0950-5849.

BARNETT, M.; LEINO, K. R. M.; SCHULTE, W. The spec# programming system: An overview. In: *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Berlin, Heidelberg: Springer-Verlag, 2005. (CASSIS'04), p. 49–69. ISBN 3-540-24287-2.

BAUERSFELD, S.; WAPPLER, S.; WEGENER, J. An approach to automatic input sequence generation for gui testing using ant colony optimization. In: . [S.l.: s.n.], 2011. p. 251–252.

BECCE, G. et al. Extracting widget descriptions from guis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 7212 LNCS, p. 347–361, 2012.

BECK. *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321146530.

BEER, A.; MOHACSI, S.; STARY, C. Idatg: an open tool for automated testing of interactive software. In: *Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International*. [S.l.: s.n.], 1998. p. 470 –475. ISSN 0730-3157.

BERTOLINI, C.; MOTA, A. A framework for gui testing based on use case design. In: *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. Washington, DC, USA: IEEE Computer Society, 2010. (ICSTW '10), p. 252–259. ISBN 978-0-7695-4050-4.

BERTOLINI, C. et al. Gui testing techniques evaluation by designed experiments. In: *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2010. (ICST '10), p. 235–244. ISBN 978-0-7695-3990-4.

BERTOLINO, A. et al. Weighting influence of user behavior in software validation. In: *Database and Expert Systems Application, 2008. DEXA '08. 19th International Workshop on*. [S.l.: s.n.], 2008. p. 495–500. ISSN 1529-4188.

BIOLCHINI, J. et al. *Systematic review in software engineering*. Rio de Janeiro/RJ - Brazil, 2005.

BLICKLE, T.; THIELE, L. A comparison of selection schemes used in evolutionary algorithms. *Evol. Comput.*, MIT Press, Cambridge, MA, USA, v. 4, n. 4, p. 361–394, dez. 1996. ISSN 1063-6560.

BOTELLA, B.; GOTLIEB, A.; MICHEL, C. Symbolic execution of floating-point computations: Research articles. *Software Testing, Verification and Reliability*, John Wiley and Sons Ltd., Chichester, UK, v. 16, n. 2, p. 97–121, 2006. ISSN 0960-0833.

BOUSQUET, L. d.; ZUANON, N. An overview of lutess: A specification-based tool for testing synchronous software. In: *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 1999. p. 208. ISBN 0-7695-0415-9.

BROOKS, P. A.; MEMON, A. M. Automated gui testing guided by usage profiles. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007. (ASE '07), p. 333–342. ISBN 978-1-59593-882-4.

BUDD, T. A. *Mutation Analysis: Ideas, Example, Problems and Prospects*. [S.l.]: North-Holand Publishing Company, 1981.

BUENO, P. M. S.; JINO, M. Identification of potentially infeasible program paths by monitoring the search for test data. In: *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2000. p. 209. ISBN 0-7695-0710-7.

CAMILO, J. C. G. *Um Algoritmo Auxiliar Paralelo inspirado na Fertilização in Vitro para melhorar o desempenho dos Algoritmos Genéticos*. Tese (Doutorado) — UFU, Uberlândia - MG, 2010.

CHAIM, M. L. *Depuração de programas baseada em informação de teste estrutural*. Tese (Doutorado) — DCA/FEE/UNICAMP, Campinas, SP, 2001.

CHEN, W.-K.; SHEN, Z.-W. Gui test-case generation with macro-event contracts. In: *Software Engineering and Data Mining (SEDM), 2010 2nd International Conference on*. [S.l.: s.n.], 2010. p. 145 –151.

CHU, H.-D.; DOBSON, J. E.; LIU, I.-C. Fast: a framework for automating statistics-based testing. *Software Quality Control*, Kluwer Academic Publishers, Hingham, MA, USA, v. 6, n. 1, p. 13–36, 1997. ISSN 0963-9314.

CHUANG, K. C.; SHIH, C. S.; HUNG, S. H. User behavior augmented software testing for user-centered gui. In: *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*. New York, NY, USA: ACM, 2011. (RACS '11), p. 200–208. ISBN 978-1-4503-1087-1.

COBERTURA, J. T. *Cobertura – A Free JAVA Tool*. 2016. Página WEB. Disponível on-line: http://cobertura.github.io/cobertura/. Acesso em: 01-02-2016.

COPELAND, L. *A Practitioner's Guide to Software Test Design*. Norwood, MA, USA: Artech House, Inc., 2003. ISBN 158053791X.

COWARD, P. A review of software testing. *Infomation and Software Technology*, v. 30, n. 3, p. 189–198, abr. 1988.

CUNHA, M. et al. Pettool: A pattern-based gui testing tool. In: *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*. [S.l.: s.n.], 2010. v. 1, p. V1–202 –V1–206.

DAVID, B. *Selenium 2 Testing Tools: Beginner's Guide*. [S.l.]: Packt Publishing, 2012. ISBN 1849518300, 9781849518307.

DELAHAYE, M.; BOUSQUET, L. Selecting a software engineering tool: Lessons learnt from mutation analysis. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 45, n. 7, p. 875–891, jul. 2015. ISSN 0038-0644. Disponível em: <http://dx.doi.org/10.1002/spe.2312>.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao teste de software*. [S.l.]: Elsevier, 2007.

DELAMARO, M. E.; MALDONADO, J. C.; MATHUR, A. P. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, v. 27, n. 3, p. 228–247, mar. 2001.

DEMILLO, R. A. Mutation analysis as a tool for software quality assurance. In: *COMPSAC80*. Chicago: [s.n.], 1980.

DEMILLO, R. A. *Software Testing and Evaluation*. [S.l.]: The Benjamin/Cummings Publishing Company Inc„ 1987.

DÍAZ, E.; TUYA, J.; BLANCO, R. Automated software testing using a metaheuristic technique based on tabu search. In: *ASE'03: Proceedings of the 18th IEEE international conference on Automated software engineering*. [S.l.: s.n.], 2003. p. 310–313.

DIJKSTRA, E. W. *Notes on Structured Programming*. The Netherlands, 1970.

DORIGO, M.; GAMBARDELLA, L. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, v. 1, n. 1, p. 53–66, 1997. ISSN 1089-778X.

DYBA, T.; DINGSOYR, T.; HANSSEN, G. K. Applying systematic reviews to diverse study types: An experience report. In: *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*. Washington, DC, USA: IEEE Computer Society, 2007. (ESEM '07), p. 225–234. ISBN 0-7695-2886-4.

EDVARDSSON, J. A survey on automatic test data generation. In: *Second Conference on Computer Science and Engineering in Linköping – ECSEL'99*. [S.l.: s.n.], 1999. p. 21–28.

EDVARDSSON, J.; KAMKAR, M. Analysis of the constraint solver in una based test data generation. In: *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, USA: ACM, 2001. p. 237–245. ISBN 1-58113-390-1.

EDWARDS, S. H. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, John Wiley and Sons, v. 11, n. 2, p. 97–111, jun. 2001.

EMBARCADERO. *Software Engineering by Automated Search SEBASE*. 2014. Página WEB. Disponível on-line: https://www.embarcadero.com/br/products/cbuilder. Acesso em: 10-02-2016.

EMER, M. C. F. P.; VERGILIO, S. R. Selection and evaluation of test data based on genetic programming. *Software Quality Control*, Kluwer Academic Publishers, Hingham, MA, USA, v. 11, n. 2, p. 167–186, 2003. ISSN 0963-9314.

ENGSTRÃM, E.; RUNESON, P. Software product line testing a systematic mapping study. *Information and Software Technology*, v. 53, p. 2–13, 2011. ISSN 0950-5849.

ENGSTRöM, E.; SKOGLUND, M.; RUNESON, P. Empirical evaluations of regression test selection techniques: a systematic review. In: ROMBACH, H. D.; ELBAUM, S. G.; MüNCH, J. (Ed.). *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, Germany*. [S.l.]: ACM, 2008. p. 22–31. ISBN 978-1-59593-971-5.

FERREIRA, L. P.; VERGILIO, S. R. Tdsgen: An environment based on hybrid genetic algorithms for generation of test data. In: *In Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO*. [S.l.]: Springer, 2004. p. 1431–1432.

FEWSTER, M.; GRAHAM, D. *Software test automation: effective use of test execution tools*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999. ISBN 0-201-33140-3.

GALLAGHER, L.; OFFUTT, J.; CINCOTTA, A. Integration testing of object-oriented components using finite state machines: Research articles. *Software Testing, Verification and Reliability*, John Wiley and Sons Ltd., Chichester, UK, v. 16, n. 4, p. 215–266, 2006. ISSN 0960-0833.

GALLAGHER, M. J.; NARASIMHAN, V. L. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 23, n. 8, p. 473–484, 1997. ISSN 0098-5589.

GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201157675.

GOLDBERG, D. E.; VOESSNER, S. Optimizing global-local search hybrids. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. [S.l.]: Morgan Kaufmann, 1999. p. 220–228.

GOTLIEB, A.; BOTELLA, B.; RUEHER, M. Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 23, n. 2, p. 53–62, 1998. ISSN 0163-5948.

GOURAUD, S.-D. et al. A new way of automating statistical testing methods. In: *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2001. p. 5.

GRILO, A.; PAIVA, A.; FARIA, J. Reverse engineering of gui models for testing. In: *Information Systems and Technologies (CISTI), 2010 5th Iberian Conference on*. [S.l.: s.n.], 2010. p. 1–6.

GUPTA, N.; MATHUR, A. P.; SOFFA, M. L. Automated test data generation using an iterative relaxation method. In: *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 1998. p. 231–244. ISBN 1-58113-108-9.

GUPTA, N.; MATHUR, A. P.; SOFFA, M. L. Una based iterative test data generation and its evaluation. In: *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 1999. p. 224. ISBN 0-7695-0415-9.

HACKNER, D. R.; MEMON, A. M. Test case generator for guitar. In: *Companion of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008. (ICSE Companion '08), p. 959–960. ISBN 978-1-60558-079-1.

HAJNAL, A.; FORGáCS, I. An applicable test data generation algorithm for domain errors. In: *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 1998. p. 63–72. ISBN 0-89791-971-8.

HARMAN, M. *Software Engineering by Automated Search SEBASE*. 2006. Página WEB. Disponível on-line: http://sebase.cs.ucl.ac.uk/. Acesso em: 10-02-2016.

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 45, n. 1, p. 11:1–11:61, dez. 2012. ISSN 0360-0300.

HARROLD, M. J. Testing: a roadmap. In: *ICSE'00: Proceedings of the Conference on The Future of Software Engineering*. New York, NY, EUA: ACM Press, 2000. p. 61–72. ISBN 1-58113-253-0.

HAUPT, R. L.; HAUPT, S. E. *Practical Genetic Algorithms*. second. New York, NY, USA: Wiley-Interscience, 2004. ISBN 0-471-45565-2.

HAYAT, M.; QADEER, N. Intra component gui test case generation technique. In: *ICIET – Information and Emerging Technologies, 2007*. [S.l.: s.n.], 2007. p. 1 –5.

HOFFMAN, D.; STROOPER, P. A.; WHITE, L. J. Boundary values and automated component testing. *Software Testing, Verification and Reliability*, v. 9, n. 1, p. 3–26, 1999.

HOU, Y.; CHEN, R.; DU, Z. Automated gui testing for j2me software based on fsm. In: *Proceedings of the 2009 International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing*. Washington, DC, USA: IEEE Computer Society, 2009. (SCALCOM-EMBEDDEDCOM '09), p. 341–346. ISBN 978-0-7695-3825-9.

HOWDEN, W. E. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8, n. 4, p. 371–379, jul. 1982.

HOWDEN, W. E. *Functional Program Testing and Analysis*. New York, NY, EUA: McGrall-Hill, 1987. ISBN 0-070-30550-1.

HOWDEN, W. E. *Software Engineering and Technology: Functional Program Testing and Analysis*. New York: McGrall-Hill Book Co, 1987.

HUANG, S.; COHEN, M. B.; MEMON, A. M. Repairing gui test suites using a genetic algorithm. In: *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2010. (ICST '10), p. 245–254. ISBN 978-0-7695-3990-4.

HUANG, Y.; LU, L. Apply ant colony to event-flow model for graphical user interface test case generation. *IET Software*, IEEE, v. 6, n. 1, p. 50–60, 2012.

IEEE. Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, p. 1–418, Dec 2010.

JENG, B.; FORGáCS, I. An automatic approach of domain test data generation. *The Journal of Systems and Software*, Elsevier Science Inc., New York, NY, USA, v. 49, n. 1, p. 97–112, 1999. ISSN 0164-1212.

KENNEDY, J.; EBERHART, R. Particle swarm optimization. In: *Neural Networks – IEEE International Conference*. [S.l.: s.n.], 1995. v. 4, p. 1942–1948 vol.4.

KHOR, S.; GROGONO, P. Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically. In: *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2004. p. 346–349. ISBN 0-7695-2131-2.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. *Science*, v. 220, p. 671–680, 1983.

KITCHENHAM, B. *Procedures for Performing Systematic Reviews*. Keele, Staffs, ST5 5BG, UK, 2004.

KITCHENHAM, B. et al. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Keele, Staffs, ST5 5BG, UK, 2007.

KOREL, B. Automated software test data generation. *IEEE Transactions on Software Engineering*, v. 16, n. 8, p. 870–879, ago. 1990.

KOREL, B. Automated test data generation for programs with procedures. In: *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 1996. p. 209–215. ISBN 0-89791-787-1.

KOREL, B.; AL-YAMI, A. M. Automated regression test generation. In: *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 1998. p. 143–152. ISBN 0-89791-971-8.

KRACHT, J. S.; PETROVIC, J. Z.; WALCOTT-JUSTICE, K. R. Empirically evaluating the quality of automatically generated and manually written test suites. In: *2014 14th International Conference on Quality Software*. [S.l.: s.n.], 2014. p. 256–265. ISSN 1550-6002.

KUK, S. H.; KIM, H. S. Automatic generation of testing environments for web applications. In: *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 02*. Washington, DC, USA: IEEE Computer Society, 2008. (CSSE '08), p. 694–697. ISBN 978-0-7695-3336-0.

KULL, A. Automatic gui model generation: State of the art. In: *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*. [S.l.: s.n.], 2012. p. 207–212.

LARRANAGA, P. et al. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artif. Intell. Rev.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 13, n. 2, p. 129–170, abr. 1999. ISSN 0269-2821.

LI, H. et al. An ontology-based approach for gui testing. In: *33rd Annual IEEE International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2009. (COMPSAC '09), p. 632–633. ISBN 978-0-7695-3726-9.

LI, H. et al. Using ontology to generate test cases for gui testing. *Int. J. Comput. Appl. Technol.*, Inderscience Publishers, Inderscience Publishers, Geneva, SWITZERLAND, v. 42, n. 2/3, p. 213–224, fev. 2011. ISSN 0952-8091.

LIN, J.-C.; YEH, P.-L. Automatic test data generation for path testing using gas. *Information Sciences: an International Journal*, Elsevier Science Inc., New York, NY, USA, v. 131, n. 1-4, p. 47–64, 2001. ISSN 0020-0255.

LINNENKUGEL, U.; MüLLERBURG, M. Test data selection criteria for (software) integration testing. In: *Proceedings of the first international conference on systems integration on Systems integration '90*. Piscataway, NJ, USA: IEEE Press, 1990. (ISCI '90), p. 709–717. ISBN 0-8186-9027-5.

LIU, X. et al. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In: *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005. p. 337–341. ISBN 1-59593-993-4.

LU, Y. et al. Development of an improved gui automation test system based on event-flow graph. In: *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 02*. Washington, DC, USA: IEEE Computer Society, 2008. (CSSE '08), p. 712–715. ISBN 978-0-7695-3336-0.

MACHADO, P. D. L.; VINCENZI, A. M. R.; MALDONADO, J. C. Ii pernambuco school on software engineering: Software testing. In: ____. 1. ed. New York, NY: Springer Berlin Heidelberg, 2010. (Lecture Notes in Computer Science, v. 6153), Incs Software Testing: An Overview, p. 1–17.

MAFRA, S. N.; TRAVASSOS, G. H. Técnicas de leitura de software: Uma revisão sistemática. In: *XIX SBES*. [S.l.: s.n.], 2005.

MAHMOOD, S. *A Systematic Review of Automated Test Data Generation Techniques*. Dissertação (Dissertação de Mestrado) — School of Engineering – Blekinge Institute of Technology, Ronneby, Sweden, out. 2007.

MALDONADO, J. C. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese (Doutorado) — DCA/FEE/UNICAMP, Campinas, SP, jul. 1991.

MALDONADO, J. C. et al. *Introdução ao Teste de Software*. [S.l.], 2004.

MANSOUR, N.; SALAME, M. Data generation for path testing. *Software Quality Control*, Kluwer Academic Publishers, Hingham, MA, USA, v. 12, n. 2, p. 121–136, 2004. ISSN 0963-9314.

MARIANI, L. et al. Autoblacktest: Automatic black-box testing of interactive applications. *Software Testing, Verification, and Validation, 2008 International Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 81–90, 2012.

MASIERO, P.; MALDONADO, J.; BOAVENTURA, I. A reachability tree for statecharts and analysis of some properties. *Information and Software Technology*, v. 36, n. 10, p. 615 – 624, 1994. ISSN 0950-5849.

MCCABE, T. J. A complexity measure. In: *Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976. (ICSE '76), p. 407–427.

MCMINN, P. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, John Wiley and Sons Ltd., Chichester, UK, v. 14, n. 2, p. 105–156, 2004. ISSN 0960-0833.

MCMINN, P. et al. The species per path approach to searchbased test data generation. In: *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2006. p. 13–24. ISBN 1-59593-263-1.

MEINKE, K.; WALKINSHAW, N. Model-based testing and model inference. In: *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change - Volume Part I*. Berlin, Heidelberg: Springer-Verlag, 2012. (ISoLA'12), p. 440–443. ISBN 978-3-642-34025-3.

MEIRELES, S. R. A. *Evolução da Ferramenta Web GUITAR para Geração Automática de Casos de Teste de Interface para Aplicações Web*. Dissertação (Mestrado) — Universidade Federal do Amazonas, 2015.

MEMON, A.; POLLACK, M.; SOFFA, M. Using a goal-driven approach to generate test cases for guis. In: *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. [S.l.: s.n.], 1999. p. 257 –266. ISSN 0270-5257.

MEMON, A. M. *A comprehensive framework for testing graphical user interfaces*. Tese (Doutorado) — University of Pittsburgh, 2001.

MEMON, A. M. An event-flow model of gui-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, John Wiley and Sons Ltd., Chichester, UK, v. 17, n. 3, p. 137–157, set. 2007. ISSN 0960-0833.

MEMON, A. M. et al. The first decade of gui ripping: Extensions, applications, and broader impacts. In: LäMMEL, R.; OLIVETO, R.; ROBBES, R. (Ed.). *WCRE*. [S.l.]: IEEE Computer Society, 2013. p. 11–20. ISBN 978-1-4799-2931-3.

MEMON, A. M.; POLLACK, M. E.; SOFFA, M. L. Hierarchical gui test case generation using automated planning. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 27, n. 2, p. 144–155, fev. 2001. ISSN 0098-5589.

MEMON, A. M.; POLLACK, M. E.; SOFFA, M. L. Hierarchical gui test case generation using automated planning. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 27, n. 2, p. 144–155, fev. 2001. ISSN 0098-5589.

MESBAH, A.; DEURSEN, A. van; LENSELINK, S. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, ACM, New York, NY, USA, v. 6, n. 1, p. 3:1–3:30, mar. 2012. ISSN 1559-1131.

MEUDEC, C. Atgen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability*, v. 11, n. 2, p. 81–96, 2001.

MIAO, Y.; YANG, X. An fsm based gui test automation model. In: *Control Automation Robotics Vision (ICARCV), 2010 11th International Conference on*. [S.l.: s.n.], 2010. p. 120–126.

MIAO, Y.; YANG, X. An fsm based gui test automation model. In: *Control Automation Robotics Vision (ICARCV), 2010 11th International Conference on*. [S.l.: s.n.], 2010. p. 120–126.

MICHAEL, C.; MCGRAW, G. Automated software test data generation for complex programs. In: *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 1998. p. 136. ISBN 0-8186-8750-9.

MICHAEL, C. C.; MCGRAW, G.; SCHATZ, M. A. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 27, n. 12, p. 1085–1110, 2001. ISSN 0098-5589.

MICHAEL, C. C. et al. Genetic algorithms for dynamic test data generation. In: *ASE '97: Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*. Washington, DC, USA: IEEE Computer Society, 1997. p. 307. ISBN 0-8186-7961-1.

MICHALEWICZ, Z. *Genetic Algorithms + Data Structures = Evolution Programs (3rd Ed.)*. London, UK, UK: Springer-Verlag, 1996. ISBN 3-540-60676-9.

MOREIRA, R.; PAIVA, A.; MEMON, A. A pattern-based approach for gui modeling and testing. In: *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. [S.l.: s.n.], 2013. p. 288–297.

MOREIRA, R. M.; PAIVA, A. C. A gui modeling dsl for pattern-based gui testing paradigm. In: *Evaluation of Novel Approaches to Software Engineering (ENASE), 2014 International Conference on*. [S.l.: s.n.], 2014. p. 1–10.

MOREIRA, R. M.; PAIVA, A. C. Pbgt tool: An integrated modeling and testing environment for pattern-based gui testing. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2014. (ASE '14), p. 863–866. ISBN 978-1-4503-3013-8.

MYERS, G. J.; SANDLER, C. *The Art of Software Testing*. [S.l.]: John Wiley & Sons, 2004. ISBN 0471469122.

NABUCO, M.; PAIVA, A. C. R. Computational science and its applications – iccsa 2014: 14th international conference, guimarães, portugal, june 30 – july 3, 2014, proceedings, part vi. In: ____. Cham: Springer International Publishing, 2014. cap. Model-Based Test Case Generation for Web Applications, p. 248–262. ISBN 978-3-319-09153-2.

NETO, A. C. D.; TRAVASSOS, G. H. A picture from the model-based testing area: Concepts, techniques, and challenges. *Advances in Computers*, v. 80, p. 45–120, 2010.

NGUYEN, B. N. et al. Guitar: An innovative tool for automated testing of gui-driven software. *Automated Software Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 21, n. 1, p. 65–105, mar. 2014. ISSN 0928-8910.

OFFUT, A. J.; LIU, S. Generating test data from sofl specifications. *The Journal of Systems and Software*, Elsevier Science Inc., New York, NY, USA, v. 49, n. 1, p. 49–62, 1999. ISSN 0164-1212.

OFFUTT, A. J.; PAN, J. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, John Wiley and Sons, v. 7, n. 3, p. 165–192, set. 1997.

OSMAN, H.; KELLY, J. *Metaheuristics: Theory and Applications*. 1. ed. [S.l.]: Kluwer Academic Publishers, 1996.

PARGAS, R. P.; HARROLD, M. J.; PECK, R. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, John Wiley and Sons, v. 9, n. 4, p. 263–282, 1999.

PENG, X.; LU, L. A new approach for session-based test case generation by ga. In: *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*. [S.l.: s.n.], 2011. p. 91 –96.

PETERSEN, K. et al. Systematic mapping studies in software engineering. In: *Proceedings of the 12th international conference on Evaluation and Assessment in Software Engineering*. Swinton, UK, UK: British Computer Society, 2008. (EASE'08), p. 68–77.

PIMENTA, A. C. R. P. *Automated Specification-Based Testing of Graphical User Interfaces*. Dissertação (Mestrado) — Engineering Faculty of Porto University, 2007.

PRESSMAN, R. S.; MAXIM, B. *Software Engineering : A Practitioner's Approach*. 8th edition. ed. [S.l.]: McGraw-Hill Science/Engineering/Math, 2014.

PRETSCHNER, A. Model-based testing. In: *Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: ACM, 2005. (ICSE '05), p. 722–723. ISBN 1-58113-963-2.

QIAN, S.; JIANG, F. An event interaction structure for gui test case generation. In: *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*. [S.l.: s.n.], 2009. p. 619 –622.

RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for program test data selection. In: *6th International Conference on Software Engineering*. Tokio, Japan: [s.n.], 1982. p. 272–278.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11, n. 4, p. 367–375, abr. 1985.

RAUF, A. et al. Automated gui test coverage analysis using ga. In: *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations*. Washington, DC, USA: IEEE Computer Society, 2010. (ITNG '10), p. 1057–1062. ISBN 978-0-7695-3984-3.

RAUF, A. et al. Pso based test coverage analysis for event driven software. In: *Software Engineering and Data Mining (SEDM), 2010 2nd International Conference on*. [S.l.: s.n.], 2010. p. 219–224.

RAUF, A.; JAFFAR, A.; SHAHID, A. Fully automated gui testing and coverage analysis using genetic algorithms. *International Journal of Innovative Computing, Information and Control*, v. 7, n. 6, p. 3281–3294, 2011.

REZENDE, S.; PRATI, R. *Sistemas Inteligentes - Fundamentos e Aplicações*. 1. ed. Barueri - SP: Manole, 2005.

ROPER, M. *Software Testing*. [S.l.]: McGrall Hill, 1994.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. [S.l.]: Prentice-Hall, 2003. (Englewood Cliffs, NJ, chapter Informed Search and Exploration). Pp. 94-131.

SANTOS-NETO, P.; RESENDE, R.; PáDUA, C. Requirements for information systems model-based testing. In: *Proceedings of the 2007 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2007. (SAC '07), p. 1409–1415. ISBN 1-59593-480-4.

SHAHZAD, A. et al. Automated optimum test case generation using web navigation graphs. In: *ICET - Internacional Conference Emerging Technologies, 2009.* [S.l.: s.n.], 2009. p. 427–432.

SILVA, C. E.; CAMPOS, J. C. Combining static and dynamic analysis for the reverse engineering of web applications. In: *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York, NY, USA: ACM, 2013. (EICS '13), p. 107–112. ISBN 978-1-4503-2138-9.

SILVA, J. L.; CAMPOS, J. C.; PAIVA, A. C. R. Model-based user interface testing with spec explorer and concurtasktrees. *Electron. Notes Theor. Comput. Sci.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 208, p. 77–93, abr. 2008. ISSN 1571-0661.

SOFFA, M. L.; MATHUR, A. P.; GUPTA, N. Generating test data for branch coverage. In: *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2000. p. 219–227. ISBN 0-7695-0710-7.

SOMMERVILLE, I. *Engenharia de Software*. 8. ed. São Paulo, SP: Addison Wesley, 2007. 568 p.

SOUZA, H. A. de. *Depuração de Programas Baseada em Cobertura de Integração*. Dissertação (Mestrado) — EACH/USP, São Paulo - SP, 2012.

SOUZA, S. R. S. *Avaliação do Custo e Eficácia do Critério Análise de Mutantes na Atividade de Teste de Programas*. Dissertação (Mestrado) — ICMC/USP, São Carlos - SP, jun. 1996.

SY, N. T.; DEVILLE, Y. Automatic test data generation for programs with integer and float variables. In: *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2001. p. 13.

SY, N. T.; DEVILLE, Y. Consistency techniques for interprocedural test data generation. In: *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2003. p. 108–117. ISBN 1-58113-743-5.

TAYLOR, B. J.; CUKIC, B. Evaluation of regressive methods for automated generation of test trajectories. In: *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2000. p. 97. ISBN 0-7695-0807-3.

Review of Four Modern Evolutionary Algorithms, v. 1. [S.l.]: EDIS - Publishing Institution of the University of Zilina, 2012. 672–677 p. ISBN 978-80-554-0551-3. ISSN 1339-9977.

TOUGAN, V.; DALOUGLU, A. e. T. An improved genetic algorithm with initial population strategy and self-adaptive member grouping. *Comput. Struct.*, Pergamon Press, Inc., Elmsford, NY, USA, v. 86, n. 11-12, p. 1204–1218, jun. 2008. ISSN 0045-7949.

TRACEY, N.; CLARK, J.; MANDER, K. Automated program flaw finding using simulated annealing. In: *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 1998. p. 73–81. ISBN 0-89791-971-8.

TRACEY, N. et al. An automated framework for structural test-data generation. In: *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 1998. p. 285. ISBN 0-8186-8750-9.

UTTING, M.; LEGEARD, B. *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN 0123725011, 9780080466484.

VEANES, M. et al. Formal methods and testing. In: HIERONS, R. M.; BOWEN, J. P.; HARMAN, M. (Ed.). Berlin, Heidelberg: Springer-Verlag, 2008. cap. Model-based Testing of Object-oriented Reactive Systems with Spec Explorer, p. 39–76. ISBN 3-540-78916-2.

VERGILIO, S. R.; MALDONADO, J. C.; JINO, M. Introdução ao teste de software. In: ____. [S.l.]: Campus, 2007. cap. Geração de Dados de Teste, p. 269–291.

VINCENZI, A. M. R. *Subídios para o Estabelecimento de Estratégias de Teste Baseadas na Técnica de Mutação*. Dissertação (Mestrado) — ICMC/USP, São Carlos - SP, nov. 1998.

VISVANATHAN, S.; GUPTA, N. Generating test data for functions with pointer inputs. In: *ASE'02: Proceedings of the 17th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2002. p. 149. ISBN 0-7695-1736-6.

WEYUKER, E. J. The complexity of data flow criteria for test data selection. *Inf. Process. Lett.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 19, n. 2, p. 103–109, set. 1984. ISSN 0020-0190.

WHITE, L.; ALMEZEN, H. Generating test cases for gui responsibilities using complete interaction sequences. In: *Proceedings of the 11th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2000. (ISSRE '00), p. 110–. ISBN 0-7695-0807-3.

WOODWARD, M. R.; HALEWOOD, K. From weak to strong, dead or alive? an analysis of some mutation testing issues. In: *Second Workshop on Software Testing, Verification and Analysis*. Banff, Canadá: [s.n.], 1988.

XIAOCHUN, Z. et al. A test automation solution on gui functional test. In: *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*. [S.l.: s.n.], 2008. p. 1413–1418. ISSN 1935-4576.

XIE, Q.; MEMON, A. Using a pilot study to derive a gui model for automated testing. *ACM Transactions on Software Engineering and Methodology*, v. 18, n. 2, 2008.

YANG, C.-S. D.; SOUTER, A. L.; POLLOCK, L. L. All-du-path coverage for parallel programs. In: *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 1998. p. 153–162. ISBN 0-89791-971-8.

YUAN, X.; COHEN, M.; MEMON, A. Towards dynamic adaptive automated test generation for graphical user interfaces. In: *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*. [S.l.: s.n.], 2009. p. 263 –266.

YUAN, X.; COHEN, M.; MEMON, A. M. Covering array sampling of input event sequences for automated gui testing. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007. (ASE '07), p. 405–408. ISBN 978-1-59593-882-4.

YUAN, X.; COHEN, M. B.; MEMON, A. M. Gui interaction testing: Incorporating event context. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 37, n. 4, p. 559–574, jul. 2011. ISSN 0098-5589.

YUAN, X.; MEMON, A. M. Using gui run-time state as feedback to generate test cases. In: *Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007. (ICSE '07), p. 396–405. ISBN 0-7695-2828-7.

YUAN, X.; MEMON, A. M. Generating event sequence-based test cases using gui runtime state feedback. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 36, n. 1, p. 81–95, jan. 2010. ISSN 0098-5589.

YUAN, X.; MEMON, A. M. Iterative execution-feedback model-directed gui testing. *Inf. Softw. Technol.*, ACM, Newton, MA, USA, v. 52, n. 5, p. 559–575, maio 2010. ISSN 0950-5849.

YUAN, X.; MEMON, A. M. Iterative execution-feedback model-directed gui testing. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 52, n. 5, p. 559–575, maio 2010. ISSN 0950-5849.

ZHU, H.; HALL, P. A. V.; MAY, J. H. R. Software unit test coverage and adequacy. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 29, n. 4, p. 366–427, dez. 1997. ISSN 0360-0300.