UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Geração de casos de teste para aplicações Web baseados em modelo de tarefas

Flavio Rezende de Jesus

Itajubá, novembro de 2015

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Flavio Rezende de Jesus

Geração de casos de teste para aplicações Web baseados em modelo de tarefas

Dissertação submetida ao Programa de Pós-Graduaçáo em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciências em Ciência e Tecnologia da Computação

Área de Concentração: Sistemas de Computação

Orientador: Prof. Dr. Laércio A. Baldochi Júnior

Novembro de 2015
Itajubá - MG

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Flavio Rezende de Jesus

Geração de casos de teste para aplicações Web baseados em modelo de tarefas

Dissertação aprovada por banca examinadora em 09 de Novembro de 2015, conferindo ao autor o título de ***Mestre em Ciências em Ciência e Tecnologia da Computação.***

Banca Examinadora:
Prof. Dr. Laércio Augusto Baldochi Júnior (Orientador)
Prof. Dr. Elisa Yumi Nakagawa
Prof. Dr. Adler Diniz de Souza

Itajubá 2015

Agradecimentos

À Deus, pela oportunidade e capacidade de realizar este trabalho, por me amparar nos momentos difíceis e me dar forças para continuar frente aos obstáculos.

Ao Prof. Dr. Laércio Augusto Baldochi Júnior, pelos conhecimentos transmitidos, pela orientação durante o desenvolvimento deste trabalho, pela oportunidade concedida e por contribuir para o meu crescimento científico, o meu profundo agradecimento.

Ao colega Leandro Guarino de Vasconcelos, por me conceder a ideia e suporte necessários para iniciar essa jornada.

À ambos, Leandro e Prof. Dr. Laércio, especialmente pelo apoio e colaboração na publicação e apresentação de nosso artigo. Não foi fácil e, com certeza, sem vocês seria impossível.

Aos meus colegas, Jonatas Zanin e Gabriel Matos que, muito gentilmente, atenderam ao meu convite e se dispuseram a doar seu tempo e seus conhecimentos, contribuindo sobremaneira com os resultados deste trabalho, especialmente por disponibilizar as ferramentas utilizadas nos experimentos.

Ao Inatel Competence Center e à Ericsson Telecomunicações S/A, pela flexibilidade e incentivos durante o desenvolvimento deste trabalho.

Aos meus pais, pelo exemplo de vida, incentivo e motivação, por investir em meus estudos e acreditar nos meus sonhos.

À todos os meus amigos e familiares, pela torcida calorosa e constante.

À minha esposa Fabiana, por estar ao meu lado em todos os momentos, de forma plena, integral e incondicional.

E finalmente, aos meus filhos, Gabriel e Daniel, que me inspiram a enfrentar os desafios e me recompensam com amor e carinho os momentos que estive ausente durante o desenvolvimento deste trabalho. Sem eles, a vida não teria o mesmo brilho.

O futuro pertence àqueles que acreditam na beleza de seus sonhos

Eleanor Roosevelt

Resumo

O tempo de desenvolvimento de *software* foi reduzido com a criação de novas ferramentas e paradigmas de programação. Além disso, a necessidade por sistemas cada vez mais eficientes e inovadores, diminuiu o tempo de lançamento entre as versões. A fim de entregar produtos de *software* em tempo hábil, assim como garantir a qualidade, segurança e corretude das aplicações Web, é imperativo a utilização de automação de testes.

Diversas técnicas foram reportadas na literatura, porém, uma maneira mais eficaz para automatizar a validação dos requisitos funcionais de um sistema Web consiste em utilizar modelo de tarefas para gerar casos de teste. Mesmo assim, as soluções baseadas nessa abordagem geralmente são custosas na geração do modelo e falham na criação dos cenários.

Para resolver os problemas reportados, foi desenvolvido o UsaTasker++, um sistema destinado à mutação de casos de teste para validação das regras de negócio das aplicações Web. O modelo proposto permite a geração dos cenários de teste usando uma abordagem simples e intuitiva, a partir da configuração e processamento do grafo correspondente ao modelo de tarefas. Para cada caso de teste gerado, o UsaTasker++ cria o respectivo *script* de teste automatizado, permitindo a sua execução e indicando os cenários com erro.

Palavras-chave: Geração de Casos de Teste. Teste automatizado. Teste baseado em modelo de tarefas.

Abstract

The software development time has been reduced with the development of new tools and programming paradigms. Besides that, the increasing need for efficient and innovative systems has reduced the time between releases. In order to deliver software products in a timely manner, as well as to guarantee quality, security and correctness of the Web applications, it is imperative to use test automation.

Several techniques has been reported, but a more effective way to automate the validation of the functional requirements of a Web system consists in using a task model to generate test cases. Even so, the solutions based on this approach are generally costly to build the model and they fail to create the scenarios.

To tackle these problems, UsaTasker++ was developed – a test case mutation system to validate the business rules of a Web application. The proposed model is capable of generating test scenarios using a simple and intuitive approach, based on the manipulation and processing of the graph related to the task model. For each test case generated, UsaTasker++ creates the corresponding automated test script, providing the capability of executing it and indicating the scenarios with errors.

Keywords: Test case generation. Automated Tests. Test based on task model.

Sumário

Lista de Figuras

Lista de Tabelas

Glossário

CSS	*Cross-Site Scripting* ou *Cascading Style Sheet*
DFS	*Depth-First Search*
DOM	*Document Object Model*
FSM	*Finite State Machine*
GUI	*Graphical User Interface*
HTML	*HyperText Markup Language*
IFML	*Interaction Flow Modeling Language*
LIFO	*Last In, First Out*
OMG	*Object Management Group*
SBSE	*Search-Based Software Engineering*
SUT	*System Under Test*
UML	*Unified Modeling Language*
XHTML	*eXtensible HTML*
XML	*eXtensible Markup Language*
WebML	*Web Modeling Language*

1 Introdução

Desde a criação da *World Wide Web* no início dos anos 90, a utilização de aplicações Web vem aumentando rapidamente. De fato, é uma das áreas que mais crescem no âmbito de sistemas de *software* (ARORA; SINHA, 2012). Tais aplicações causaram um grande impacto em diversos aspectos da nossa sociedade, desde negócios, educação, governo, setores de entretenimento, indústria, às nossas vidas pessoais. Apesar do grande envolvimento dessas aplicações no dia a dia da comunidade, ainda há muito o que evoluir no quesito qualidade, e para isso, o suporte de ferramentas de testes automatizados é essencial.

Conforme estudo realizado pela Forbes (FORBES, 2014), das empresas que sofreram danos em suas imagens ou no valor de suas marcas durante o ano de 2013 e 2014, 66% tiveram danos devido a falhas nos sistemas de tecnologia da informação. Elbaum et al. (ELBAUM *et al.*, 2005) também referenciam pesquisas que apontam altos índices de falhas em sistemas de comércio eletrônico e sites governamentais. Por esses motivos, testes de aplicações, principalmente de aplicações Web, tornaram-se parte crítica do processo de desenvolvimento de *software*.

Os testes de sistemas ainda são atividades predominantemente manuais, repetitivas e, por vezes, artesanais. Papadakis et al. (PAPADAKIS; MALEVRIS; KALLIA, 2010) afirmam que tais atividades podem consumir 50% ou até mesmo 60% do custo do ciclo de vida do *software*. Além disso, a operação humana pode demandar mais tempo e causar muitos erros.

De fato, os testes automatizados são considerados cruciais para o sucesso de grandes aplicações Web (LEOTTA *et al.*, 2013). Uma das principais vantagens da automação é permitir aos desenvolvedores de *software* a capacidade de executar os testes com mais frequência e encontrar os *bugs* nos estágios iniciais de desenvolvimento. Tais mecanismos economizam muito tempo, visto que rodam mais rápidos e podem ser executados durante a noite. Por fim, a automação também ajuda a lançar aplicações com menos erros.

Para demonstrar a ausência de erros em uma aplicação, seria necessário testar todas as permutações para todas as situações possíveis. Porém, para sistemas não-triviais, tal abordagem exaustiva é tecnicamente e economicamente inviável. Na realidade, não é possível verificar todas as possibilidades até mesmo para um sistema trivial (LI; DAS; DOWE, 2014).

Uma das formas de garantir que um sistema computacional atenda aos seus requisitos dentro de um determinado nível de qualidade é através da utilização de técnicas de teste de *software*. Os testes baseados em modelos são uma das técnicas mais referenciadas na verificação de sistemas computacionais (CHUANG; SHIH; HUNG, 2011). Nesta abordagem, o comportamento e a estrutura de um sistema são representados de maneira precisa e não ambígua por um modelo formal. Os casos de teste são, então, gerados com base no modelo construído (LI; DAS; DOWE, 2014).

Normalmente, cada requisito de um sistema computacional possui um ou mais casos de teste a ele associado. Shirole e Kumar (SHIROLE; KUMAR, 2013) definem os casos de teste como um conjunto de passos para executar um teste, operando sob um conjunto de entradas pré-definidas para produzir um resultado esperado. Neste contexto, um conjunto de casos de teste deve verificar a funcionalidade geral do sistema – se o mesmo está em conformidade com o documento de especificação ou se expõe falhas de *software* (e.g., erros de funcionalidade ou segurança).

Apesar das diversas técnicas e ferramentas apresentadas na literatura, ainda há carência por metodologias robustas, capazes de realizar validações na maioria dos sistemas Web existentes de forma simples, completa e ágil. Não obstante, o senso de urgência na liberação das versões de *software* só aumenta este desafio.

No sentido de mitigar os problemas citados, este trabalho apresenta o UsaTasker++, uma ferramenta para gerar todos os cenários de teste possíveis referentes a uma tarefa. O modelo de representação, criado durante a gravação da tarefa no USABILICS (VASCONCELOS; BALDOCHI JR., 2011), é convertido em um grafo direcionado, o qual é processado para gerar todos os casos de teste. Além disso, a ferramenta é capaz de converter os casos em testes automatizados, executar os testes criados e sinalizar os resultados de sucesso ou com erros.

No primeiro instante, a ferramenta USABILICS é utilizada para gravar as ações executadas em uma aplicação Web para, então gerar o grafo direcionado correspondente. Como as ações são executadas diretamente na interface gráfica do sistema sendo avaliado, tal passo é simples e rápido. Posteriormente, o UsaTasker++ utiliza um conjunto

de algoritmos e técnicas para processar o grafo e, em alguns segundos, gerar os casos de teste relacionados. Por fim, a ferramenta utiliza bibliotecas renomadas no mercado para mapear cada caso de teste em testes automatizados e executá-los.

A validação e os resultados apresentados neste trabalho sugerem que a metodologia e a ferramenta UsaTasker++ são eficazes na geração e execução de casos de teste capazes de validar os requisitos funcionais de uma aplicação Web.

1.1 Objetivos

A fim de contribuir para a pesquisa relacionada a validações práticas e eficientes de aplicações Web e também para suprir as deficiências das abordagens reportadas na literatura, este trabalho teve como objetivo desenvolver um sistema para auxiliar no processo de geração de casos de teste e descoberta dos possíveis erros presentes nessas aplicações.

No sentido de alcançar este objetivo geral, os seguintes objetivos específicos foram delineados:

1. permitir a criação de todos os casos de teste relacionados a uma tarefa base;
2. permitir a execução automática dos casos gerados e a indicação dos cenários com problemas;
3. contribuir para a evolução do USABILICS (VASCONCELOS; BALDOCHI JR., 2011, 2012a) como uma plataforma robusta e mais completa para testes de aplicações Web;

1.2 Metodologia

As pesquisas desenvolvidas na área de geração de casos de teste para aplicações Web (LI; DAS; DOWE, 2014; LUCCA; FASOLINO, 2006; ALALFI; CORDY; DEAN, 2009; KAM; DEAN, 2009; DOGAN; BETIN-CAN; GAROUSI, 2014; BAU *et al.*, 2010; ARORA; SINHA, 2012; MYERS; SANDLER, 2004; IVORY; HEARST, 2001; RICCA; TONELLA, 2001) constatam a necessidade de uma abordagem que facilite tanto a geração quanto a execução dos cenários, a fim de reduzir os custos e tempo gastos na etapa de testes, sem diminuir a qualidade dos sistemas implementados. No intuito de suprir

essa necessidade, essa seção descreve os métodos utilizados para alcançar os objetivos específicos delineados neste trabalho.

Para alcançar o objetivo 1, foi desenvolvida uma metodologia composta por cinco passos, os quais mesclam algoritmos de processamento de grafos com técnicas da álgebra elementar para gerar todos os cenários possíveis de uma determinada tarefa. Através dessa metodologia, foi possível implementar uma ferramenta chamada UsaTasker++, que permite ao avaliador definir tarefas diretamente na interface da aplicação, configurar o modelo de tarefas e gerar os casos de teste para a validação completa dos requisitos funcionais. Além disso, o método implementado otimiza o tempo e reduz o esforço necessário para definição dos casos de teste.

A fim de contemplar o objetivo 2, foi implementado um módulo de automação no UsaTasker++ para converter os elementos do modelo de tarefas em comandos para manipulação de navegadores. Com isso, a ferramenta permite executar automaticamente cada caso de teste gerado e indicar os cenários com erro.

Por fim, para alcançar o objetivo 3, o UsaTasker++ foi embutido como módulo na ferramenta USABILICS (VASCONCELOS; BALDOCHI JR., 2011). Com essa transformação, a plataforma de testes passou a suportar (i) avaliação remota de usabilidade, (ii) geração automática de casos de teste e (iii) execução de testes automatizados. Além disso, tais funcionalidades são apoiadas por interfaces de configuração da própria plataforma, concentrando diversas atividades da fase de teste em apenas uma ferramenta.

O UsaTasker++, baseado na metodologia definida para processamento do grafo relativo ao modelo de tarefas, incorpora os resultados obtidos nos objetivos especificados na Seção 1.1, permitindo a geração de casos de teste e a sua execução automática.

1.3 Contribuições

As técnicas utilizadas neste trabalho foram detalhadas em (JESUS; VASCONCELOS; BALDOCHI, 2015) e implementadas no UsaTasker++, um módulo da ferramenta USABILICS. A inclusão de testes funcionais em seu escopo, tornou a ferramenta mais robusta e prática. Ressalta-se, ainda, que as soluções desenvolvidas são disponíveis para utilização por terceiros através do endereço eletrônico *https://gitlab.com/flavio-rezende/USABILICS*.

1.4 Estrutura da dissertação

A dissertação está estruturada da seguinte forma:

O Capítulo 2 descreve os principais conceitos relacionados a testes de *software*, as técnicas definidas na literatura para tratar as diversas abordagens existentes e os trabalhos relacionados a geração de casos de teste.

O Capítulo 3 apresenta o sistema UsaTasker++, detalhando sua estrutura e a técnica criada para gerar casos de teste. Este capítulo também apresenta os artefatos gerados pelo sistema USABILICS, o qual possibilitou a criação do UsaTasker++.

No Capítulo 4 são apresentados os conceitos de testes automatizados, alguns trabalhos relacionados que utilizam as técnicas de execução automática de testes e os seus benefícios e problemas. Este capítulo apresenta também as ferramentas de automação utilizadas pelo UsaTasker++.

Os experimentos realizados para análise e validação do UsaTasker++ são detalhados no Capítulo 5, demonstrando a sua eficácia na geração de casos de teste e as dificuldades encontradas para execução automática dos mesmos.

Finalmente, o Capítulo 6 apresenta as considerações finais, resumindo as principais contribuições do trabalho e apresentando sugestões para trabalhos futuros.

2 Teste de Aplicações Web

O teste de aplicações é o meio mais difundido para a garantia do cumprimento dos requisitos de um sistema de *software*. Tal processo fornece uma crítica ou balizador, a fim de comparar o estado e comportamento do produto com especificações, contratos, versões passadas, expectativas, padrões, normas, leis e outros critérios (LEITNER *et al.*, 2007).

Durante a última década, diversos pesquisadores propuseram diferentes técnicas para análise e teste de aplicações Web (DOGAN; BETIN-CAN; GAROUSI, 2014). Tais abordagens foram criadas para validar um aspecto específico do sistema, seja ele funcional ou não-funcional. Além disso, diversas ferramentas que implementam essas técnicas e geram cenários de testes foram desenvolvidas para diminuir o esforço gasto durante a fase de validação do sistema.

Este capítulo aborda as diferenças entre uma aplicação tradicional e um sistema Web, além de descrever os principais conceitos relacionados a testes funcionais e não-funcionais. Os processos descritos na literatura para validação dos *softwares* e as estratégias para geração de casos de teste também são apresentados. Por fim, discute-se sobre as ferramentas que suportam os testes automatizados e suas limitações.

2.1 Diferenças entre aplicações tradicionais e Web

Os testes de aplicações Web compartilham dos mesmos objetivos de uma aplicação tradicional: confiabilidade, usabilidade, operabilidade, segurança e corretude. Apesar disso, na maioria dos casos, as teorias e métodos de testes para sistemas convencionais não podem ser utilizados para verificação de um sistema Web dadas as suas peculiaridades e complexidades (LUCCA; FASOLINO, 2006).

De acordo com Di Lucca e Fasolino, as aplicações Web possuem uma grande quantidade de usuários distribuídos por todo o mundo acessando o sistema concorrentemente. Yuan-Fang et al. (LI; DAS; DOWE, 2014) reforçam que a utilização paralela depende de

um gerenciamento de recursos (conexão, banco de dados, arquivos, memória, etc.) efetivo, tornando difícil a reprodução e detecção de erros por ferramentas de testes. O fato de ser acessível globalmente também aumenta a complexidade dos testes, visto que o aumento da visibilidade da aplicação a torna mais suscetível a diversos ataques e, consequentemente, amplia o leque de possibilidades a serem testadas.

Outra característica definida por Di Lucca e Fasolino é o fato de sistemas Web possuírem ambientes de execução heterogêneos, com hardware, conexões, sistema operacional, servidor Web e *browser* diferentes. A variedade de tecnologias empregadas aumentam significativamente os cenários e combinações possíveis a serem verificados.

Aplicações Web também são capazes de gerar componentes de software em tempo de execução de acordo com as entradas do usuário e estado do servidor. Neste caso, o desafio das ferramentas de testes é abordar cada estado e sua respectiva combinação de entradas, levando em consideração a lógica associada ao sistema.

Yuan-Fang et al. ainda citam outras características, como o fato de aplicações Web utilizarem múltiplas linguagens de programação e tecnologias em constante evolução, obrigando as técnicas de testes a se manterem atualizadas e genéricas.

Cada aspecto descrito anteriormente produz novos desafios e perspectivas relacionadas a testes. O principal objetivo é encontrar falhas no respectivo serviço ou funcionalidade, e assim evitar que o comportamento do sistema não esteja de acordo com o esperado. Portanto, para organizar o processo de validação em busca do objetivo citado, os testes são caracterizados em dois grupos: funcionais e não-funcionais.

2.2 Testes não-funcionais

Os testes não-funcionais relacionam a conformidade do sistema com os requisitos implícitos ou explícitos, geralmente dissociados da funcionalidade em si. Eles verificam se os elementos do sistema foram adequadamente integrados, se estão de acordo com as normas, e como se comportam frente às restrições tecnológicas. Conforme (LUCCA; FASOLINO, 2006) os tipos mais comuns são:

- **Desempenho:** O objetivo do teste de desempenho é verificar o comportamento (e.g. tempo de resposta, acessibilidade) do sistema simulando centenas de acessos simultâneos em um determinado período de tempo. Falhas de desempenho geralmente estão relacionadas a insuficiência dos recursos computacionais ou má distribuição

de tais recursos.

- **Carga:** O teste de carga envolve a avaliação do funcionamento do sistema considerando determinado nível de carga. Com isso, avalia-se o tempo para realizar as funções com base na condição específica do sistema. As falhas encontradas nesse tipo de teste geralmente estão relacionadas ao sistema sob o qual a aplicação está executando.

- **Estresse:** O teste de estresse avalia o comportamento do sistema em condições superiores aos limites estabelecidos nos requisitos. O objetivo é verificar se o sistema fica corrompido e se é capaz de se recuperar de tais circunstâncias.

- **Compatibilidade:** O teste de compatibilidade está relacionado com a utilização do sistema em diferentes plataformas ou navegadores. Verifica-se as diferentes possibilidades de configuração e o funcionamento integrado das diferentes versões.

- **Usabilidade:** O foco dos testes de usabilidade é medir a facilidade de uso e operação de determinada aplicação. Esses testes geralmente estão associados a interface gráfica do sistema, se o formato visual está correto, se as informações são claras e precisas, etc.

- **Acessibilidade:** O teste de acessibilidade pode ser considerado um tipo especial de teste de usabilidade em que seu objetivo é verificar se o acesso ao conteúdo do sistema é realmente válido, mesmo em situações com configurações reduzidas ou deficiência física do usuário.

- **Segurança:** O objetivo do teste de segurança é verificar se o sistema é capaz de se defender contra o acesso não autorizado, além de preservar os recursos do sistema da má utilização. Para isso, o teste de segurança busca confirmar que as pessoas autorizadas possuem acesso aos serviços e recursos autorizados.

Algumas ferramentas foram desenvolvidas especificamente para execução de testes não-funcionais. Gias e Sakib (GIAS; SAKIB, 2014) propuseram uma abordagem bayesiana para selecionar determinadas URLs e efetuar verificações intensas na aplicação com o intuito de testar o seu desempenho. Vasconcelos e Baldochi (VASCONCELOS; BALDOCHI JR., 2011) apresentaram outra abordagem cujo resultado foi a implementação do USABILICS – um sistema voltado à avaliação remota e semiautomática de usabilidade baseada em um modelo de interface. Já Lionel et al. (BRIAND; LABICHE; SHOUSHA, 2005) utilizaram algoritmos genéticos para derivar os casos de teste e aumentar o tempo

de resposta de determinadas operações com o objetivo de estressar o sistema. No âmbito de segurança, Yang et al. (YANG *et al.*, 2009) desenvolveram um modelo baseado em lógica difusa para gerar entradas aleatórias e observar os resultados apresentados.

2.3 Testes funcionais

Os testes funcionais estão relacionados com a verificação dos serviços e suas respectivas regras de negócio implementadas no sistema. De acordo com Di Lucca e Fasolino (LUCCA; FASOLINO, 2006), os testes de função baseiam-se em (i) modelos, que representam as relações entre elementos e componentes de software, (ii) processos, que definem o fluxo das atividades de testes a serem executadas e seu gerenciamento, (iii) níveis, que especificam o escopo dos testes e componentes a serem testados, e (iv) estratégias, que definem a metodologia e algoritmos para criar os casos de teste.

Os itens previamente listados endereçam diferentes problemas no contexto de aplicações Web, conforme detalhado a seguir.

2.3.1 Modelos de representação

No âmbito de testes de software, os modelos representam conceitos ou relacionamentos essenciais sobre os itens a serem testados. Tais modelos podem ser utilizados para suportar a seleção de casos de teste, determinando comportamentos específicos ou focando em aspectos estruturais do software.

Os modelos tradicionais, como o UML (*Unified Modeling Language*), são deficientes para expressar os comportamentos específicos de aplicações Web (ROSSI, 2013). Tais modelos não conseguem representar navegação, composição de páginas e comportamentos interativos. Já os modelos criados para a verificação de aplicações Web são extensões dos modelos tradicionais, provendo funcionalidades extras para representar explicitamente as características de softwares para a Web. O IFML (*Interaction Flow Modeling Language*) é o modelo padrão para aplicações Web adotado pelo OMG[1] (*Object Management Group*) – organização internacional responsável por aprovar padrões abertos para aplicações orientadas a objetos.

A Figura 1 descreve um exemplo simples de modelo IFML, no qual o usuário pode realizar a busca por um produto entrando com os critérios no formulário de Busca de

[1] www.omg.org

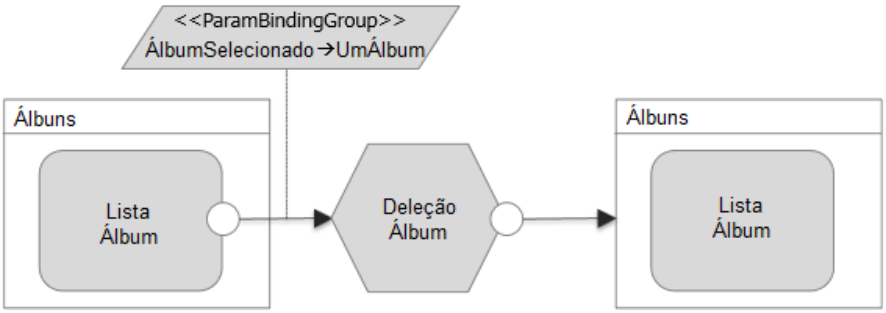


Figura 1: Exemplo IFML: busca por produto, listagem e deleção.

Produtos. Os itens resultantes da pesquisa são exibidos em uma lista, conforme o retângulo do lado esquerdo. A seleção de um item (paralelogramo) acarreta em sua remoção (hexágono), que, ao final, atualiza a lista a ser exibida (retângulo direito). A sequência de ações é representada pelas setas no modelo.

Além do IFML, outros modelos exclusivos para aplicações Web foram discutidos na literatura. Os modelos *Comportamentais* descrevem o comportamento da aplicação independentemente dos detalhes de implementação. Tais modelos suportam os testes de caixa-preta, onde o foco é no comportamento externo, sem considerar as técnicas utilizadas para chegar no resultado esperado. Já os modelos *Estruturais* baseiam-se nos conceitos e tecnologias envolvidas para o desenvolvimento do sistema, suportando os testes de caixa-branca, onde os detalhes de implementação internos são avaliados exaustivamente.

2.3.2 Processos de testes

O processo para execução dos testes pode ser manual ou automatizado. Nos testes manuais, um analista de testes executa uma sequência de atividades e verifica se o sistema se comporta conforme esperado. Esse tipo de processo geralmente é utilizado para tarefas intelectuais e que exigem uma análise e pensamento lógico. Por outro lado, os testes manuais demandam um grande esforço e são mais suscetíveis a erros.

Já os testes automatizados utilizam ferramentas de software que executam os testes no sistema sem a intervenção humana, comparando os resultados esperados com os resultados reais. Para que isso seja possível, é necessário configurar as pré-condições, as funções de controle e os relatórios dos testes. Dado essa carga de configuração e preparação, esse processo não é recomendado para sistemas com ciclo de vida curto. Sistemas grandes e em constante modificação utilizam tal processo para garantir que uma alteração no sistema não interfira em outras partes não desejadas. No geral, a automação é utilizada para

testes repetitivos e que precisam ser executados com frequência. Ela é mais rápida que os testes manuais e menos suscetível a erros.

2.3.3 Níveis de testes

O processo de teste de software define um conjunto de atividades para serem executadas em diferentes níveis do ciclo de desenvolvimento do software. Os três principais níveis são listados a seguir.

- Teste de Unidade: Os testes unitários geralmente são os primeiros dessa cadeia de atividades, e neles, cada pedaço de código (método, componente ou módulo) é verificado com base nos valores de entrada e no comportamento de saída. Tais testes são concentrados em descobrir os erros dentro do escopo da unidade. Eles focam na lógica interna de processamento e nas estruturas de dados dentro dos limites de um componente.

- Teste de Integração: Já os testes de integração consideram as diversas partes do sistema e verificam como elas trabalham juntas. O foco desses testes é a validação das interfaces das unidades e das informações que transitam entre elas. O esforço aqui é na validação do comportamento do sistema, e não nas unidades específicas.

- Teste de Regressão: Cada vez que as unidades de software são modificadas ou novas unidades são incluídas no sistema, novos comportamentos podem surgir. Os testes de regressão verificam o funcionamento do sistema como um todo para garantir que nenhum módulo ou integração de módulos foram afetados de forma indesejada. Os testes básicos são reexecutados a fim de validar que a parte do sistema que não sofreu alteração continua funcionando da mesma forma que antes. Geralmente os testes de regressão são executados por testes automatizados.

2.3.4 Estratégias de testes

As estratégias de testes definem as abordagens para desenvolvimento dos casos de teste. Cada abordagem determina um ponto de referência para criação dos casos de teste, os critérios de avaliação, as entradas e saídas, critérios de parada dos testes e o objetivo principal. A seguir, as principais estratégias são detalhadas.

Teste Baseado em Grafos e Modelos A abordagem de testes baseados em grafos utiliza modelos que representam a aplicação Web, conforme detalhado na seção 2.3.1. O modelo é então traduzido em um grafo ou em uma máquina de estado que determina as possíveis configurações e estados do sistema. Por fim, os casos de teste são criados seguindo uma das seguintes técnicas: ou todas as diretivas são verificadas e testadas ou todos os caminhos são percorridos e testados (LI; DAS; DOWE, 2014).

Qualquer objeto envolvendo pontos com conexões entre eles pode ser chamado de grafo. Os modelos baseados em grafos representam páginas Web como vértices e os links entre elas como arestas. A estratégia de testes para esse tipo de modelo é percorrer as arestas e verificar se todas as páginas são alcançáveis, se existe erro na sequência de páginas ou se as lógicas dos caminhos mais curtos e longos estão corretas (SCIASCIO *et al.*, 2002; ALFARO; HENZINGER; MANG, 2001).

As técnicas baseadas em grafos geralmente utilizam DOM – *Document Object Model*. Conforme proposto por Ricca e Tonella (MARCHETTO; TONELLA; RICCA, 2008; RICCA; TONELLA, 2001), cada vértice no grafo deste modelo representa um objeto Web (como páginas, formulários, quadros e botões) e as arestas representam relações entre esses objetos (como enviar, incluir, dividir e conectar). Tais definições são uma convenção independente de plataforma e de linguagem de desenvolvimento para referenciar e interagir com objetos em documentos HTML, XHTML e XML. Para a execução dos testes gerados basta disparar as possíveis ações nos objetos mapeados.

Já a abordagem de geração de testes baseados em máquinas de estados finitos (*Finite State Machine*, FSM) considera a relação entre o estado de uma aplicação Web e o seu comportamento neste estado. A partir desse modelo, os testes esperam soluções específicas para endereçar as explosões dos estados (LUCCA; FASOLINO, 2006). Conforme Yuan-Fang et al. (LI; DAS; DOWE, 2014), a abordagem de máquinas de estados é amplamente utilizada em testes de aplicações Web pois tais sistemas essencialmente contemplam as transições de uma página para outra, além de produzirem saídas com base nas ações, entradas e estado corrente.

A Figura 2, apresentada em (ANDREWS; OFFUTT; ALEXANDER, 2005), exemplifica o uso de FSM para uma aplicação Web simples com campos de usuário e senha. Neste exemplo, a página só pode ser enviada se os campos *Usuário* e *Senha* estiverem preenchidos, independente da ordem de preenchimento dos mesmos.

Teoricamente, aplicações Web podem ser completamente modeladas utilizando Máquinas de Estados Finitos. Porém, até mesmo uma página simples pode sofrer com o

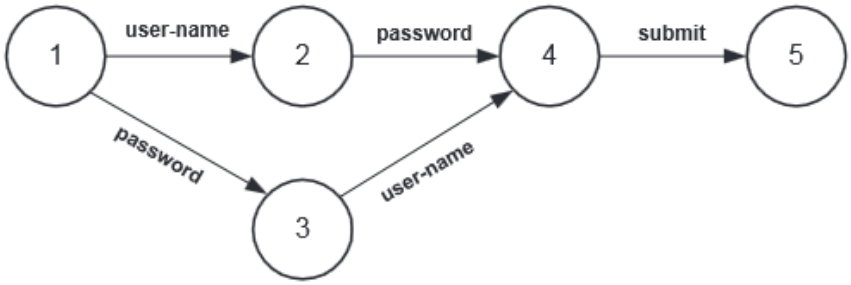


Figura 2: Exemplo de FSM para uma aplicação Web

problema de explosão de espaço dos estados. Ou seja, um número imenso de possibilidades inviabilizaria o modelo para geração de casos de teste (ANDREWS; OFFUTT; ALEXANDER, 2005).

Outra abordagem amplamente utilizada para criação de casos de teste é o modelo baseado em tarefas. A tarefa é uma atividade que deve ser executada para alcançar determinado objetivo na aplicação. Utilizado como um artefato de análise de requisito e desenho de software, o modelo de tarefas captura conhecimento sobre as ações que o sistema suporta e as camadas de lógica envolvidas, descrevendo as hipóteses de como o usuário interage com o *software* (BARBOSA; PAIVA; CAMPOS, 2011). Para gerar os casos de teste, basta explorar tais hipóteses, como, por exemplo, modificando as entradas do usuário ou a ordem com que as ações são executadas.

A Figura 3 mostra um Modelo de Tarefas para a compra de um produto num sistema de *e-commerce* definido em (VASCONCELOS; BALDOCHI JR., 2012b). Neste exemplo, o objetivo da tarefa é realizar a compra do produto na aplicação. Para isso, o usuário deve percorrer as ações conforme detalhadas no modelo: abrir a página *DEALS*, selecionar a operação *DEAL*, entrar com os detalhes, até completar o pedido (*COMPLETE ORDER*).

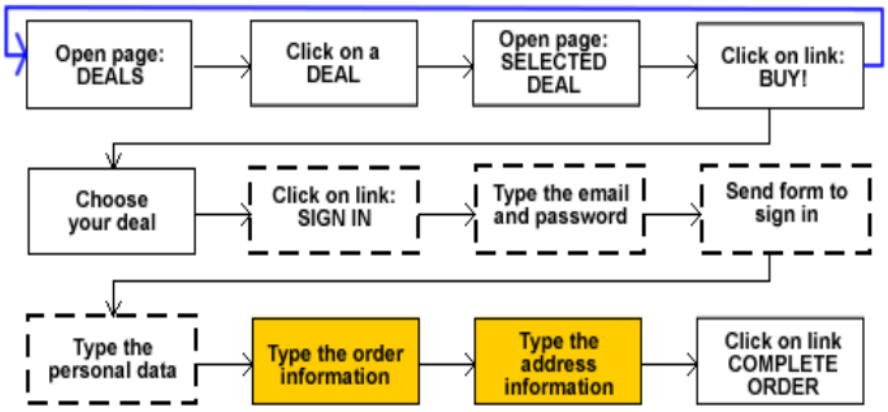


Figura 3: Exemplo de Modelo de Tarefas

Teste de Mutação Teste de Mutação é uma estratégia de validação de sistemas baseada em falhas. Esta abordagem produz alterações sintáticas no modelo que representa o código sob teste com o intuito de revelar falhas semânticas na aplicação (PAPADAKIS; MALEVRIS; KALLIA, 2010). Os testes gerados a partir do modelo mutante pretendem introduzir erros típicos de usuários do sistema. Geralmente, tal abordagem é utilizada em conjunto com outros modelos de grafos.

Essa técnica é considerada como teste negativo, no qual o principal objetivo é revelar uma brecha no sistema para corromper o funcionamento tradicional do *software*. Teoricamente, o sistema deve se proteger contra aqueles cenários que não são considerados como "caminhos felizes", e, assim, evitar que o *software* seja abortado.

Neste contexto, Barbosa et al. (BARBOSA; PAIVA; CAMPOS, 2011) criaram testes apenas reordenando os eventos do usuário e verificando como o sistema se comportava. Eles também expandiram a técnica habilitando e desabilitando os componentes do sistema em um estado específico.

Teste Baseado em Busca A estratégia de testes baseados em Busca é derivada da Engenharia de Software Baseada em Busca (*Search-Based Software Engineering*, SBSE) definida por (HARMAN; JONES, 2001). Tal abordagem utiliza técnicas de meta-heurísticas, como algoritmos genéticos, busca tabu e arrefecimento simulado para resolver problemas de otimização. Em alguns casos, a complexidade computacional do sistema é tão grande que não é possível adotar meios tradicionais para solucionar um problema, portanto, as técnicas de otimização são recomendadas.

Alshahwan e Harman (ALSHAHWAN; HARMAN, 2011) explicam que, primeiro, as possíveis soluções precisam ser codificadas de tal maneira que seja possível obter resultados aproximados dentro de um espaço de busca. Depois, uma função de avaliação precisa ser definida para comparar as soluções. Por fim, operadores que alteram uma solução sem sucesso precisam ser selecionados de tal modo que haja um direcionamento para outra solução melhor. O resultado dessa abordagem é a produção de um conjunto de testes que maximizam a cobertura de verificação sob a aplicação.

Estratégias de Varredura e Crawling Vulnerabilidades de segurança representam sérios riscos para as aplicações Web. Em muitas aplicações, essas vulnerabilidades resultam de validações de entradas genéricas, o que permite ataques de *SQL injection* e *Cross-Site Scripting* (CSS) (LI; DAS; DOWE, 2014). As técnicas de varredura são capazes

de identificar esses problemas injetando entradas inválidas no sistema e determinando que tipos de erros existem conforme o comportamento apresentado. Já os *Crawlers* são ferramentas que navegam pela página Web e coletam informações de maneira predefinida e automatizada (BRETON; MARONNAUD; HALLÉ, 2013). Tanto a varredura quanto os *crawlers*, além de serem muito eficientes nos testes para detecção de vulnerabilidades, também são utilizados para análise de desempenho (BAU *et al.*, 2010).

Teste Randômico Teste Randômico é uma estratégia simples e muito conhecida em testes de sistemas, na qual os testes são gerados a partir de entradas aleatórias para a aplicação (GODEFROID; KLARLUND; SEN, 2005). Na maioria dos casos, a cobertura dos testes que utilizam essa técnica é pequena, visto que a probabilidade de se gerar entradas aleatórias e válidas que passem por todas as alternativas do código é pequena. Por outro lado, essa técnica pode ser muito útil para sistemas que não necessitam de verificação de grande cobertura e que não possuem recursos disponíveis para grandes validações.

Teste baseado em Sessão de Usuário Um dos grandes limitantes na criação de casos de teste é o custo para descobrir um conjunto satisfatório de valores de entrada capaz de exercitar o sistema. A abordagem dos Testes Baseados em Sessão de Usuário trabalha exatamente nessa limitação. Conforme Elbaum et al. (ELBAUM; KARRE; ROTHERMEL, 2003; ELBAUM *et al.*, 2005), a operação em um sistema Web normalmente consiste em receber e processar uma requisição. Portanto, com poucas configurações no servidor, é possível coletar de forma transparente as interações dos usuários na forma de URLs e conjuntos chave-valor. Depois, pode-se aplicar um conjunto de técnicas para utilizar as informações coletadas e gerar casos de teste.

As estratégias de testes apresentadas definem os principais conceitos necessários para a geração de casos de teste, porém, muitas vezes, aplicar as técnicas e seguir o protocolo pode demandar um esforço gigantesco de implementação. Uma das formas de automatizar as estratégias definidas e otimizar o processo de verificação do sistema é por meio de ferramentas de geração de casos de teste.

2.4 Ferramentas para geração de casos de teste

Testes manuais de software podem se tornar excessivamente tediosos, demorados e, principalmente, caros. Além disso, tais atividades estão propensas a falhas humanas, tornando a automação dos testes, por menor que seja, altamente desejada (THUMMALAPENTA *et al.*, 2013).

Analistas de testes e desenvolvedores normalmente estão sob severa pressão devido aos curtos ciclos de desenvolvimento de versões de *software*. Tal dificuldade têm encorajado as organizações a procurar por técnicas que apresentem melhorias frente à tradicional maneira artesanal de elaborar casos de teste individuais. A utilização de ferramentas que automatizam o processo de geração de casos de teste é uma alternativa vantajosa para tratar esses problemas (ALSHAHWAN; HARMAN, 2011).

Diversas técnicas e ferramentas foram apresentadas na literatura para suportar os testes em aplicações Web, mas a maioria desses estudos focaram em conformidade com protocolos de comunicação, testes de carga, detecção de links inválidos, validação de HTML, e análises estáticas que não endereçam a validação funcional do sistema (ARORA; SINHA, 2012).

O trabalho feito por Ricca e Tonella (RICCA; TONELLA, 2001) foi um dos primeiros a gerar resultados relevantes na geração de casos de teste voltados para a análise funcional de sistemas Web. Abstrações de grafos foram utilizadas para descrever a aplicação. Neste modelo, os nós representavam as páginas, formulários e quadros, e as arestas representavam as relações entre os nós, como links e requisições. Em seguida, os nós e arestas foram utilizados como critérios de cobertura para derivar os casos de teste. Uma das principais limitações deste trabalho é o fato do grafo ter que ser criado manualmente, tornando difícil automatizar o processo. Além disso, somente os caminhos lineares independentes são verificados.

Além de Ricca e Tonella, vários outros autores propuseram abordagens baseadas em modelos para geração de casos de teste. Andrews et al. (ANDREWS; OFFUTT; ALEXANDER, 2005) utilizaram Máquinas de Estados Finitos (FSM) para modelar os sistemas Web. De acordo com o modelo proposto por eles, nós na FSM representam páginas Web e as arestas representam a transição entre as páginas. A principal limitação dessa abordagem é a necessidade de criar a máquina de estados manualmente, dificultando a automação completa do processo.

Thummalapenta et al. (THUMMALAPENTA *et al.*, 2013) também exploraram di-

agramas de transição de estados, que são criados por sistemas que varrem (*crawlers*) a interface gráfica da aplicação. Neste caso, uma série de algoritmos processa o diagrama para identificar os caminhos e as regras de negócio relevantes. Por abordar regras de negócio na geração de casos de teste, este é o único estudo encontrado na literatura capaz de garantir a validade do cenário gerado. Apesar do trabalho de Thummalapenta et al. considerar tal feito, os experimentos mostraram que apenas parte das regras de negócio são efetivamente consideradas.

Já Alshahwan e Harman (ALSHAHWAN; HARMAN, 2011) utilizaram os conceitos de problemas de otimização par desenvolver o sistema SWAT . Tal sistema utiliza algoritmos genéticos para modificar os dados de entrada e gerar novos casos de teste. Apesar de eficaz, os resultados demonstraram que a abordagem é lenta quando comparada a outras técnicas.

Outros esforços também foram realizados para gerar casos de teste sem a dependência de modelos. Praphamontripong e Offutt (PRAPHAMONTRIPONG; OFFUTT, 2010) implementaram uma aplicação para testes de mutação específicos em *Java Server Pages* (JSP) e *Java Servlets*. À medida que as páginas eram exploradas, testes mutantes eram criados. A limitação dessa abordagem é o foco nos testes negativos, conforme exposto na Seção 2.3.4.

Técnicas de varredura exploram exaustivamente as páginas da aplicação e criam cenários de teste com base no grafo navegacional (BENEDIKT; FREIRE; GODEFROID, 2002; WANG *et al.*, 2009). As primeiras abordagens de varredura exploraram os *hyperlinks* tradicionais dentro de cada página. Porém, com o avanço das tecnologias Java e Ajax, novas abordagens foram propostas para considerar as mudanças de estado que não envolvem um novo carregamento da página (BRETON; MARONNAUD; HALLÉ, 2013; MESBAH; DEURSEN; LENSELINK, 2012). Apesar dessas técnicas conseguirem gerar um volume enorme de casos de teste, essa abordagem não é efetiva em cobrir todos os requisitos funcionais do sistema visto que baseiam-se no fluxo da varredura, e não na lógica das ações permitidas.

Frantzen et al. (FRANTZEN *et al.*, 2009) propuseram um sistema para testar automaticamente as especificações funcionais de *Web Services*. As entradas selecionadas aleatoriamente de uma lista de entradas eram passadas para o serviço, invocando assim, uma operação. Em seguida, a mensagem de retorno era verificada com a especificação formal para definir o resultado do teste. O principal limitante dessa abordagem é a dificuldade em definir uma lista de entradas capaz de abordar todas as especificações funcionais.

Mesmo assim, não é possível garantir que testes serão gerados para todos os casos.

As sessões de usuário também são utilizadas para geração de casos de teste. Os cenários são criados a partir dos dados coletados durante a utilização da aplicação (ELBAUM *et al.*, 2005; SPRENKLE *et al.*, 2005). As vantagens de gravar sessões de usuário são, além de aproximar os casos de teste aos cenários reais executados, também permitem capturar sequências inusitadas de ações, visto que o usuário pode ficar navegando aleatoriamente na aplicação até decidir completar a tarefa. Por outro lado, é difícil garantir a cobertura dos testes resultantes, visto que eles dependem dos dados gravados. Ainda assim, mesmo que as principais tarefas fossem executadas, não é possível garantir que todas as possibilidades e caminhos permitidos pela aplicação sejam executados.

2.5 Considerações finais

Neste capítulo, foram apresentados os principais conceitos relacionados a testes de *software*, após uma breve comparação entre sistemas tradicionais e Web. É possível perceber que, apesar de possuírem objetivos comuns, as aplicações Web provém maior abrangência, são mais heterogêneas e fornecem a maioria dos serviços disponíveis em sistemas de *software*.

No intuito de organizar o processo de validação das aplicações Web, os tipos de testes foram caracterizados em funcionais e não-funcionais. Os requisitos implícitos, como desempenho, segurança e usabilidade, os quais determinam as características complementares aos requisitos definidos do sistema, são avaliados por meio de técnicas específicas de testes. Já os testes funcionais são baseados em modelos, processos, níveis e estratégias.

Os modelos descrevem a forma como os testes representam o sistema. Já os processos estabelecem o formato entre manual ou automatizado. Os níveis relacionam as atividades a serem executadas dentre os testes de unidade, integração e regressão. Por fim, as estratégias definem as abordagens para geração dos casos de teste.

Por ser dispendiosa e propensa a falhas humanas, a verificação manual deu lugar a sistemas de automação de testes. Diversas ferramentas foram desenvolvidas para a criação de casos de teste, porém, percebeu-se uma característica em comum nas estratégias analisadas: nenhuma das ferramentas foi capaz de criar testes que validam completamente os requisitos funcionais dos sistemas. A geração automática de testes não trata os problemas de cobertura incerta do domínio.

As estratégias que utilizam modelos de representação demandam um grande esforço manual para criação dos modelos. As que utilizam técnicas de varredura não garantem a execução de todas as regras do negócio, assim como a abordagem randômica. Além disso, as estratégias baseadas em sessão de usuário são limitadas pelo escopo executado na sessão de origem.

Com o intuito de preencher essas lacunas, este trabalho concentrou esforços na criação do UsaTasker++, uma ferramenta para geração de casos de teste baseados nos requisitos funcionais do sistema. A ferramenta disponibiliza uma suíte de serviços voltados para verificação e validação de aplicações Web de forma simples, e, acima de tudo, eficaz na cobertura dos requisitos funcionais.

3 Geração de Casos de Teste Baseada em Tarefas - UsaTasker++

Durante o processo de verificação de *software*, é comum encontrar um grande número de possíveis cenários de teste, mesmo em sistemas simples. Escolher os melhores casos a serem executados é uma tarefa importante neste processo. Casos de teste devem ser projetados de tal maneira a cobrirem 100% dos requisitos funcionais e devem ser efetivos na descoberta de defeitos (SHIROLE; KUMAR, 2013).

Este capítulo aborda a ferramenta UsaTasker++, desenvolvida neste trabalho para geração de casos de teste capazes de validar os requisitos funcionais de uma aplicação Web. A seção 3.1 apresenta o USABILICS – um sistema para avaliação remota de usabilidade de sistemas Web – e como o mecanismo de gravação de tarefas desse sistema permitiu a criação de um modelo para geração de casos de teste. Por fim, a seção 3.2 detalha o modelo de tarefas e a metodologia utilizada pelo UsaTasker++ para a geração dos casos de teste.

3.1 Origens do UsaTasker++

Apesar de não estar relacionado com testes de usabilidade, este trabalho originou-se a partir das oportunidades vislumbradas nas pesquisas do USABILICS – ferramenta de suporte e análise de usabilidade de sistemas Web (VASCONCELOS; BALDOCHI JR., 2011).

3.1.1 USABILICS e o modelo de tarefas

As abordagens mais utilizadas para avaliação remota de usabilidade são baseadas em modelos de tarefas gerados a partir dos *logs* dos eventos capturados no servidor ou no cliente (IVORY; HEARST, 2001). A comparação entre a sequência de eventos realizada pelo usuário na execução de uma dada tarefa e a sequência de eventos definida pelo modelo

do *log* é capaz de indicar eventuais problemas de usabilidade.

Neste contexto, Vasconcelos e Baldochi criaram o USABILICS (VASCONCELOS; BALDOCHI JR., 2011, 2012a). O sistema provê uma interface gráfica simples e intuitiva para definição e análise de tarefas, capaz de capturar as ações do avaliador sobre os elementos da interface Web, estruturá-las em um modelo de tarefas e comparar os modelos gerados para determinar a qualidade da usabilidade do sistema.

A análise de tarefas é feita encontrando-se a similaridade entre a sequência de eventos da tarefa definida pelo avaliador (modelo de tarefas do USABILICS) e as sequências de eventos das interações dos usuários (modelo dos *logs* armazenados). Vasconcelos e Baldochi propuseram um índice de usabilidade para mensurar se as semelhanças e disparidades na comparação entre os dois modelos são resultados de um problema de usabilidade.

A proposta de um índice baseado na análise de tarefas proporciona aos avaliadores uma visualização rápida e resumida sobre a usabilidade da interface gráfica em relação às tarefas definidas, permitindo a comparação entre os valores do índice antes e depois de alguma readaptação da interface, por exemplo.

Além de determinar o índice de usabilidade, o USABILICS também sugere melhorias para corrigir ou minimizar alguns erros de usabilidade comumente encontrados em elementos da página Web ou na sequência de eventos executados pelo usuário.

Apesar da eficácia do índice de usabilidade, existem tarefas que podem apresentar sequências de eventos não lineares. Por exemplo, a tarefa *adicionar um produto no carrinho de compras* numa aplicação de *e-commerce* permite que os eventos selecionar produto, adicionar e remover do carrinho, sejam executados diversas vezes antes de finalizar a tarefa. Com isso, mesmo que a tarefa tenha sido executada da maneira esperada, a diferença entre o modelo de tarefas e o *log* será grande, prejudicando o índice de usabilidade.

Para solucionar esse problema, Vasconcelos e Baldochi desenvolveram o UsaTasker (VASCONCELOS; BALDOCHI JR., 2012b). A ideia da ferramenta é prover uma interface gráfica para configuração do modelo de tarefas do USABILICS e, assim, indicar os eventos que podem ser executados em qualquer ordem, eventos opcionais e eventos que podem ser repetidos várias vezes.

Um evento fora de ordem seria, por exemplo, o preenchimento do último nome antes do preenchimento do primeiro nome, durante o cadastro do usuário em um sistema de comércio eletrônico. Da mesma maneira, inserir objetos no carrinho de compras repetidas vezes, seria um evento repetido. Por fim, um evento opcional poderia ser considerado o

aceite ou não dos "Termos de compromisso" do sistema Web.

A Figura 4 ilustra o grafo direcionado gerado pelo UsaTasker. Os elementos fora de ordem estão com fundo amarelo e os elementos opcionais possuem contorno pontilhado. O elemento repetido é representado pela seta que liga um elemento posterior a um elemento anterior.

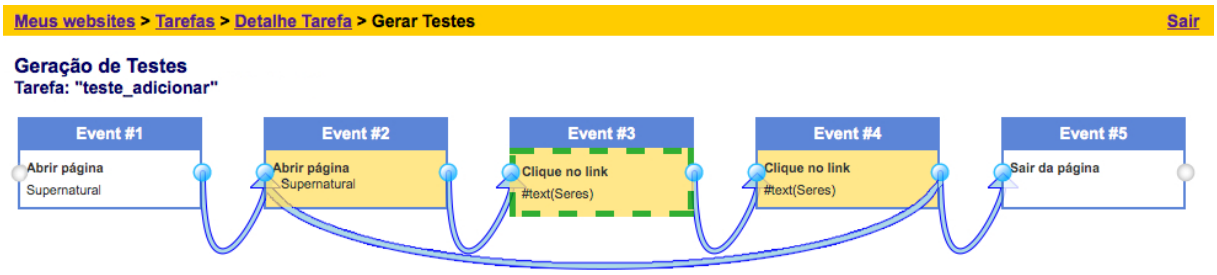


Figura 4: Um modelo de tarefa no UsaTasker

3.2 UsaTasker++

Além dos resultados apresentados pelo USABILICS em torno da avaliação de usabilidade de sistemas Web dinâmicos, notou-se que os artefatos gerados nessas pesquisas poderiam ser utilizados para outros tipos de testes. Os grafos gerados pelo sistema representam as tarefas definidas pelo usuário. Ao categorizar os elementos desses grafos como *opcionais*, *fora de ordem* e *repetidos*, outras sequências de eventos para as tarefas ficam acessíveis. Cada novo caminho disponibilizado no grafo representa uma forma de se executar corretamente a tarefa.

A Figura 5 exibe algumas das novas sequências de eventos geradas após a categorização no UsaTasker. Considerando a sequência original (A), os caminhos em (B), (C) e (D) representam novas sequências formadas pela categorização do evento como *opcional, fora de ordem* e *repetido*, respectivamente.

Notou-se que, a partir de um único cenário gravado, além da categorização dos eventos da tarefa, é possível gerar novas sequências de casos de teste para validar os requisitos funcionais do sistema. Para implementar tal generalização, a ferramenta UsaTasker foi estendida e denominada de UsaTasker++.

O objetivo deste trabalho é demonstrar que, ao explorar o modelo navegacional do UsaTasker, é possível extrair casos de teste de um grafo direcionado que representa uma tarefa. Levando em consideração que uma determinada atividade pode ser *opcional*, *fora de ordem* ou *repetida*, é possível executar uma tarefa de várias formas diferentes.

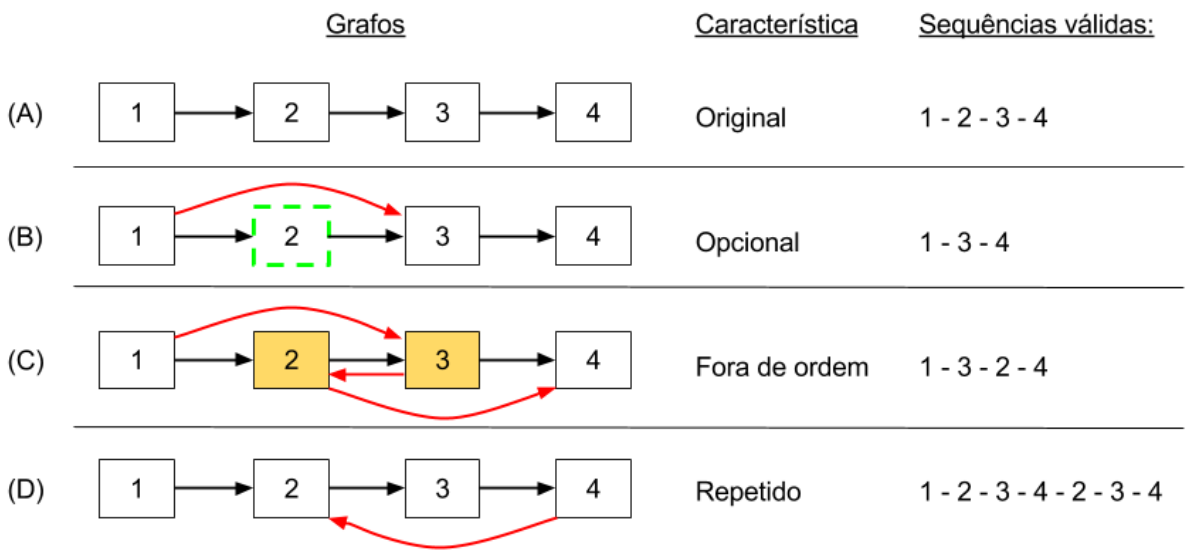


Figura 5: Novas sequências de eventos após categorização no UsaTasker

Em outras palavras, há vários caminhos do ponto inicial (primeiro evento) ao final da tarefa (evento final). O objetivo do UsaTasker++ é descobrir todos esses caminhos, transformando cada possibilidade em um caso de teste.

Para produzir todos os caminhos possíveis que possuem eventos iniciais e finais em comum, um conjunto de algoritmos foi desenvolvido para processar um grafo direcionado considerando as funcionalidades do modelo navegacional. Diferentes técnicas foram utilizadas para processar o grafo, distribuídas nos seguintes passos:

1. calibrar grafo: cria uma lista adjacente que representa o grafo básico e adiciona novas arestas;
2. descobrir caminhos: encontra todos os caminhos básicos no grafo;
3. aplicar método de redução: remove todos os caminhos inválidos;
4. processar fora de ordem: cria caminhos adicionais para representar os eventos fora de ordem;
5. processar ciclo: cria caminhos adicionais para incluir os eventos repetidos.

A Figura 6 exibe a evolução das ferramentas, desde o USABILICS aos detalhes do UsaTasker++.

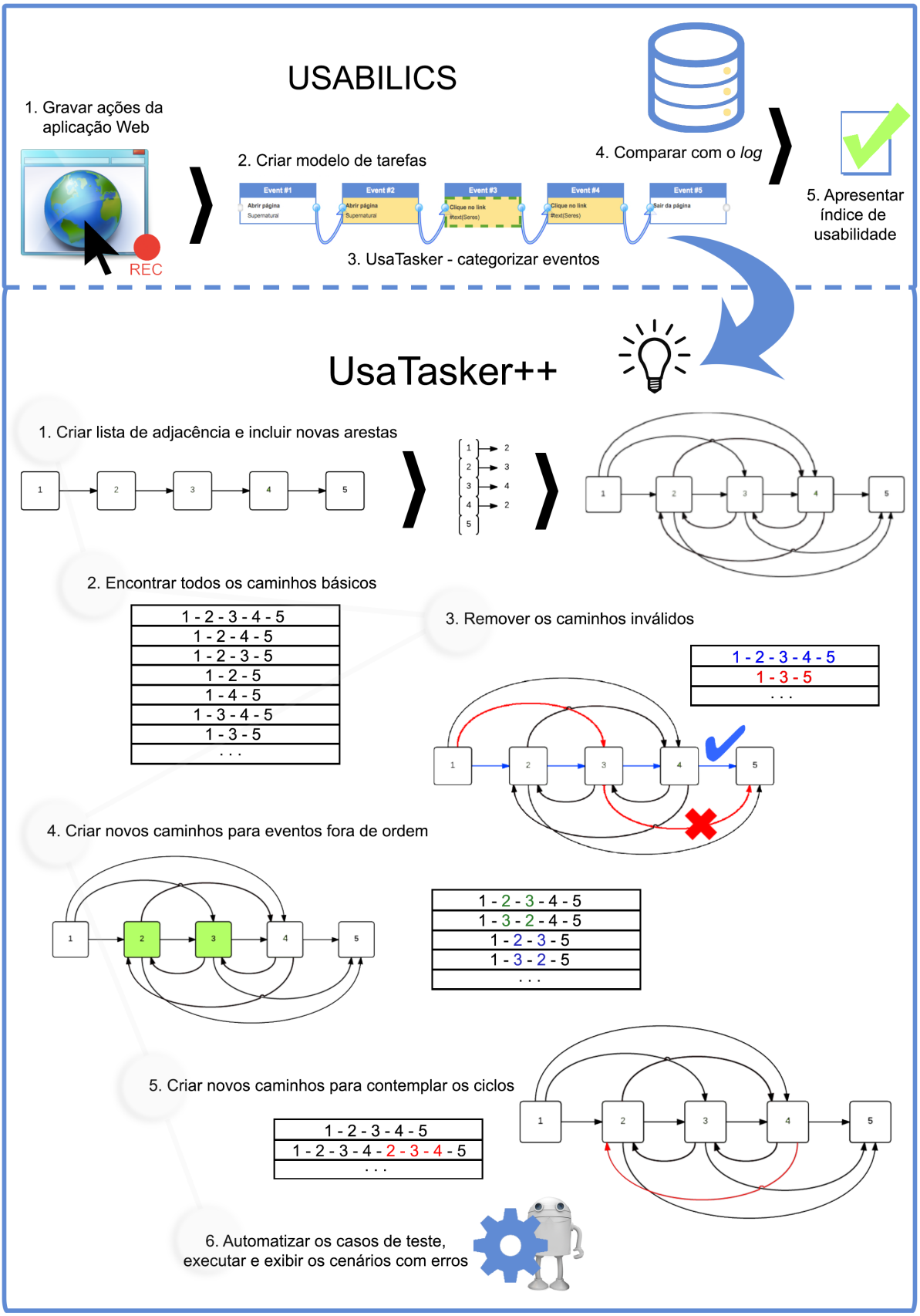


Figura 6: Os principais passos do USABILICS e do UsaTasker++

3.2.1 Passo 1: Calibrar grafo

Neste passo, o grafo original gerado pelo UsaTasker é convertido em uma lista de adjacência. Em seguida, a lista é refatorada em múltiplas iterações para endereçar de maneira efetiva as três opções de eventos: *opcional, cíclico* e *fora de ordem.*

Um evento opcional cria uma nova aresta entre o evento anterior e o evento posterior ao do vértice opcional. Neste caso, o usuário pode pular um evento, indo direto de uma ação anterior para outra posterior, conforme mostra a Figura 7 (A). Um exemplo dessa característica é uma aplicação Web na qual o usuário pode preencher opcionalmente o segundo nome ou marcar a opção de subscrever a uma notificação. A restrição para esse tipo de evento é a impossibilidade de deixar o evento inicial ou final como opcionais. Eles são mandatórios visto que os limites (inicial e final) são utilizados para determinar o início e fim do processamento do grafo.

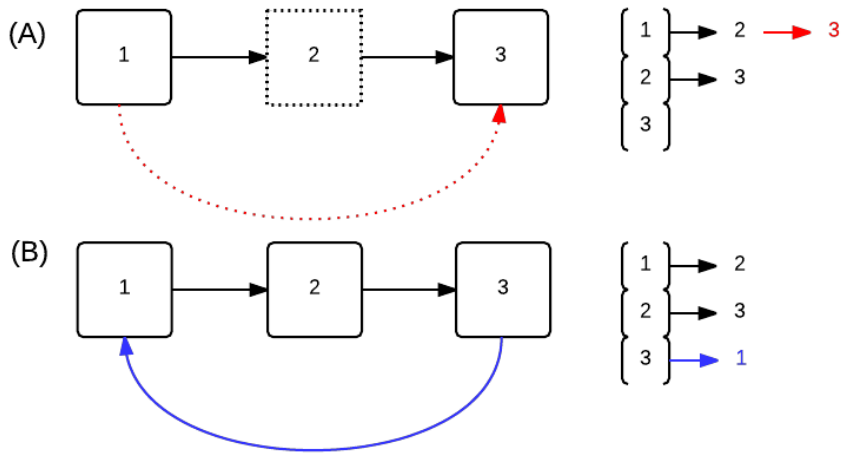


Figura 7: Exemplos de Grafos e suas Listas de Adjacências

O evento cíclico conecta um vértice a outro localizado em posição anterior no grafo. Esse tipo de evento é utilizado nos casos onde um conjunto de eventos se repetem dentro de um mesmo fluxo. A Figura 7 (B) mostra um exemplo de ciclo. Os ciclos só são processados caso o número de elementos entre os vértices seja superior a um. Essa restrição diferencia um evento fora de ordem, que pode ser representado como um ciclo entre dois vértices vizinhos. Não há restrições quanto ao nó de destino do ciclo. Este pode ser opcional e fora de ordem. Em aplicações de comércio eletrônico, as operações para adicionar múltiplos itens ao carrinho de compras consiste de ações cíclicas.

Um evento fora de ordem deve sempre coexistir com outro, gerando diversas arestas adicionais. No exemplo da Figura 8, os cinco eventos 1, 2, 3, 4 e 5 - onde 2, 3 e 4 são fora

de ordem - indicam que é possível ter diversas sequências de eventos: 1-2-3-4-5; 1-3-2-4-5; 1-4-3-2-5, e assim por diante. As quatro fases deste processamento são listadas a seguir.

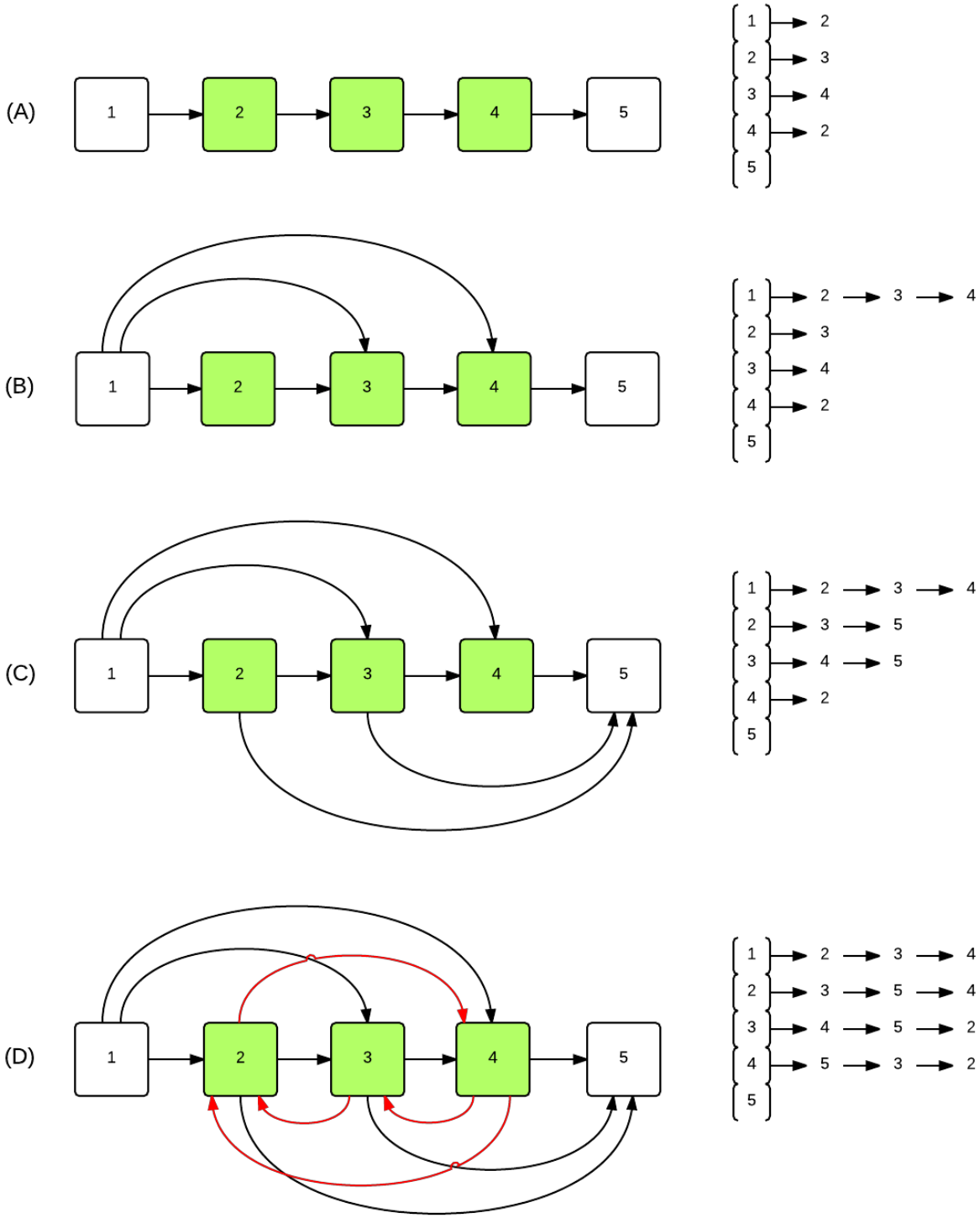


Figura 8: Grafo com Elementos Fora de Ordem

Na primeira fase – Figura 8 (A) – os elementos fora de ordem são agregados em uma lista. Um grafo pode ter diversas listas de elementos fora de ordem desde que as listas

sejam separadas por um elemento que não seja nem fora de ordem, nem opcional. Durante a segunda fase, o nó anterior a lista é conectado a todos os elementos da lista, conforme Figura 8 (B). Na terceira fase todos os elementos da lista são conectados ao primeiro nó posterior à lista – Figura 8 (C). Por fim, cada elemento da lista é conectado a todos os outros elementos – Figura 8 (D). Como exemplo, esse tipo de situação pode ocorrer em uma aplicação Web onde vários campos de texto podem ser preenchidos em qualquer ordem.

Considerando que a densidade de um grafo é calculada com base na proporção entre a quantidade de arestas e vértices, pode-se afirmar que, à medida que as fases são executadas no primeiro passo, maior a densidade do grafo.

3.2.2 Passo 2: Descobrir caminhos

O objetivo do segundo passo é gerar os caminhos que representam as diferentes sequências de eventos. Com a Lista de Adjacência modificada na etapa anterior, todos os caminhos são definidos utilizando o Algoritmo 1, também denominado de DFS Manipulado (*Depth-First Search*, Busca em Profundidade).

O que diferencia este algoritmo do DFS tradicional é a utilização de gatilhos (como na linha 5), que permite referenciar a sequência de eventos, e não somente a árvore DSF.

A pilha (LIFO) definida como L no algoritmo é utilizada para representar a sequência de eventos corrente. Toda vez que o algoritmo avança no grafo, o evento é inserido na pilha (linha 3). Quando ele retrocede ao elemento anterior, o evento é removido da pilha (linha 27). Ao executar essas ações o algoritmo está sempre em controle da sequência de eventos que representa o caminho até o nó atual.

No exemplo da Figura 9, quando o algoritmo atinge o nó 7 após o passo *d*, a pilha está completa com a sequência 1-2-3-5-7. Então, o algoritmo retorna para um nó cujo caminho ainda não foi explorado. Ele alcança o nó 2 no passo *g*, onde existe a possibilidade do algoritmo ir adiante (pelo nó *4*). Neste ponto, a pilha contém os elementos *1-2*. Portanto, essa sequência já inicia com *1-2* e no momento em que chega ao nó 7, irá conter a sequência 1-2-4-6-7.

Outra diferença em relação ao DFS tradicional é que este algoritmo reinicia o estado de *visitado* dos nós para *não visitado*. Isso acontece toda vez que o fluxo chega ao elemento final do grafo (linha 8). Isso permite que o mesmo nó seja visitado mais de uma vez, caso um caminho diferente seja capaz de alcançá-lo.

Algoritmo 1: Algoritmo *DFSManipulado*

Entrada: Evento E, lista linear de eventos L, sequência de eventos Ps, índice do caminho atual p, lista de adjacência Adj
Saída: lista de caminhos encontrados Ps, hash-map C do vértice e adjacente

```
1  begin
2      inicie E como visitado ;
3      puxe E to L ;
4      adicione E to Ps(p) ;
5      if E é o último evento no caminho then
6          incremente p ;
7          adicione nova sequência a Ps ;
8          reinicie todos em Adj para não visitados ;
9      end
10     firstAdj ← verdadeiro ;
11     foreach evento adj em Adj(E) do
12         if a > E then
13             if firstAdj é falso then
14                 foreach evento a em L do
15                     adicione evento a a Ps(p) ;
16                 end
17                 firstAdj ← falso ;
18                 if adj é não visitado then
19                     DFSManipulado(adj, L, Ps, p, Adj) ;
20                 end
21             end
22         end
23         else
24             adicione E e adj a C ;
25         end
26     end
27     remova último de L ;
28 end
```

Quando o algoritmo processa cada elemento no caminho, ele verifica se o próximo elemento está realmente à frente (linha 12), caso contrário ele irá considerar que o caminho possui um ciclo. Durante a criação do grafo, cada vértice é associado a um identificador único em ordem crescente. Dessa forma é possível garantir que um elemento mais avançado no grafo está realmente à frente de um outro elemento com identificador menor. Caso isso não ocorra, significa que o algoritmo está passando por um ciclo, então os elementos envolvidos devem ser armazenados em uma lista específica de ciclos para serem processados posteriormente (linha 24).

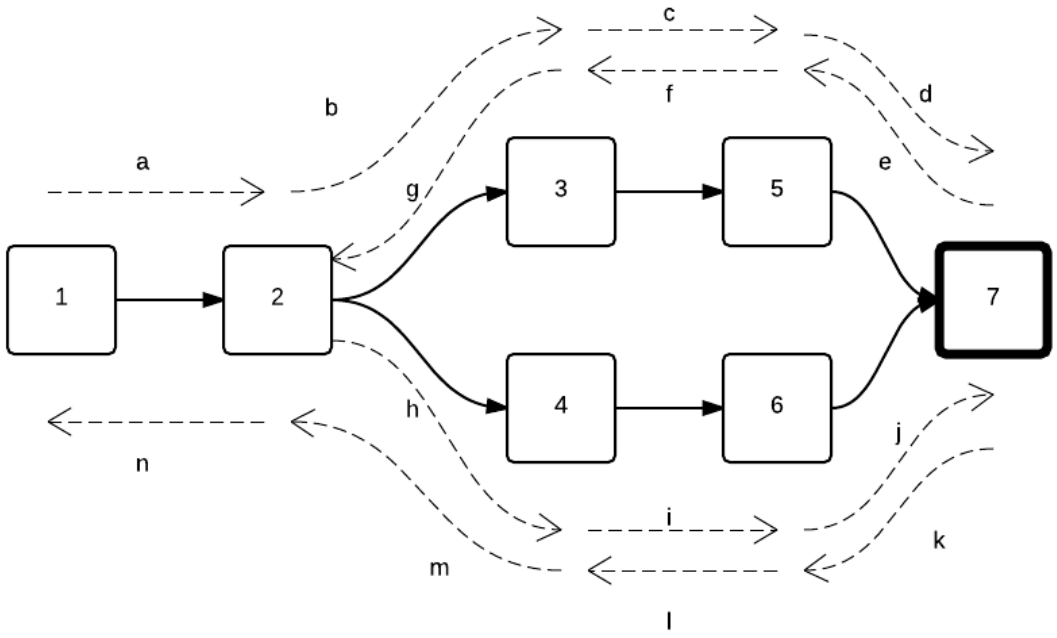


Figura 9: Execução do algoritmo DFS Manipulado.

3.2.3 Passo 3: Método de Redução

Neste ponto a Lista de Adjacência já foi criada e os caminhos definidos. Porém, a maioria dos caminhos criados durante a fase anterior são inválidos, visto que, ao processar os elementos fora de ordem, diversas arestas foram criadas sem respeitar algumas regras do sistema. Como todos os elementos de um conjunto fora de ordem são conectados entre si, alguns caminhos deixam de considerar elementos obrigatórios.

No exemplo da Figura 10, os eventos 2, 3 e 4 são fora de ordem e 3 é um evento opcional. Neste caso, o caminho 1-2-5 é aceitável para o algoritmo DSF, mas não é um cenário de teste válido por não possuir o evento 4 ("não opcional"). A Tabela 1 lista os caminhos gerados após o processamento do grafo da Figura 10.

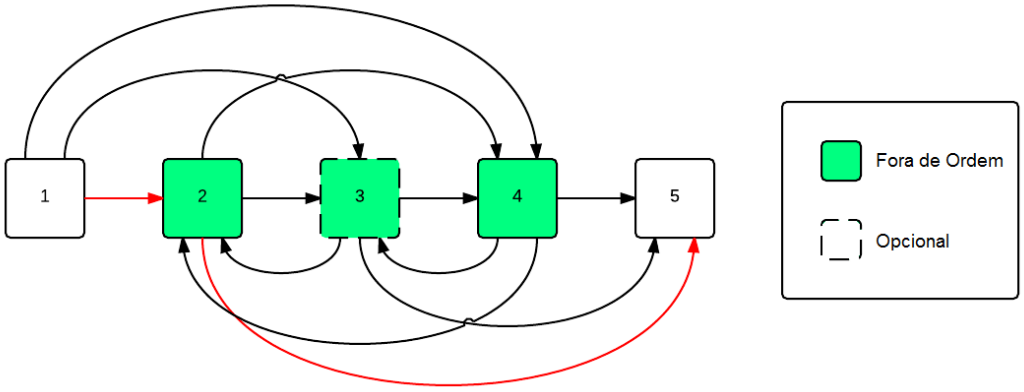


Figura 10: Caminho inválido.

Tabela 1: Validação dos caminhos

Caminho	É válido?	Caminho	É válido?
1-2-3-4-5	Sim	1-4-5	Não – faltando o 2
1-2-4-5	Sim – 3 é opcional	1-4-3-5	Não – faltando o 2
1-2-3-5	Não – faltando o 4	1-4-2-5	Sim – 3 é opcional
1-2-5	Não – faltando o 4	1-3-4-5	Não – faltando o 2
1-2-4-3-5	Sim	1-3-2-5	Não – faltando o 4
1-3-2-4-5	Sim	1-3-5	Não – faltando o 2 e 4
1-3-4-2-5	Sim	1-4-2-3-5	Sim
1-4-3-2-5	Sim		

Durante essa fase, o método de redução percorre todos os caminhos e verifica se os elementos obrigatórios ("não opcionais") estão presentes. Se houver algum elemento faltando, o caminho é desconsiderado.

3.2.4 Passo 4: Processar Fora de Ordem

O objetivo deste passo é descobrir as variações de um único caminho, baseado nos eventos fora de ordem. Caso um caminho contenha um conjunto fora de ordem, o algoritmo de Permutação (SEDGEWICK, 1977) é chamado para gerar todas as sequências possíveis que este conjunto pode ter. Em seguida, o sistema replica o caminho uma vez para cada variação, mantendo os elementos externos do conjunto e modificando as sequências.

No exemplo exibido na Figura 10, o conjunto fora de ordem 2-3-4 gera as seguintes sequências após a chamada da permutação: [2,3,4]; [2,4,3]; [3,2,4]; [3,4,2]; [4,2,3]; [4,3,2]. Os nós externos 1 e 5 são replicados em cada uma das seis variações. O resultado final dos caminhos gerados a partir do conjunto fora de ordem são exibidos na Tabela 2.

Tabela 2: Caminhos gerados

1 - 2 - 3 - 4 - 5
1 - 2 - 4 - 3 - 5
1 - 3 - 2 - 4 - 5
1 - 3 - 4 - 2 - 5
1 - 4 - 2 - 3 - 5
1 - 4 - 3 - 2 - 5

Note que neste ponto, caso algum elemento do conjunto fora de ordem seja opcional, o processo de DFS detalhado nas seções anteriores já define um caminho com o elemento opcional e outro caminho sem o elemento. Assim, a permutação é executada separada-

mente para cada caminho, considerando apenas os elementos no conjunto. No exemplo da Figura 10, a permutação é executada para o caminho 1-2-3-4-5 e depois para 1-2-4-5.

O algoritmo de permutação troca a posição dos elementos para gerar a próxima permutação. O resultado desta operação é uma lista com todas as permutações. Por exemplo, o conjunto {1,2} resulta em [1,2] e [2,1]; o conjunto {2,3,4} produz [2,3,4]; [2,4,3]; [3,2,4]; [3,4,2]; [4,2,3]; [4,3,2].

Além do processo de permutação, pode ser necessário lidar com mais de um conjunto fora de ordem em um único caminho. Caso cada conjunto fosse tratado separadamente, o algoritmo de permutação iria gerar caminhos variando somente os elementos de dentro daquele conjunto e iria ignorar os outros conjuntos fora de ordem no caminho. Para tratar este cenário, o Algoritmo 2 - *ProcessarForaDeOrdem* - foi criado e é capaz de executar a permutação entre mais de um conjunto fora de ordem em um mesmo caminho.

Algoritmo 2: Algoritmo *ProcessarForaDeOrdem*

```
   Entrada: lista P de eventos do caminho
   Saída: lista N de novos caminhos
 1 begin
 2     permMap ← mapOOOSequences(P) ;
 3     foreach caminho p em permMap do
 4         foreach sequência fora de ordem ooo em p do
 5             permGroup ← permutação(ooo) ;
 6             numPerm ← numPerm× número de permutações em
               permutação(ooo) ;
 7         end
 8         for iterador i=0 to número de permGroup do
 9             for iterador index=0 to numPerm do
10                 for iterador k to permGroup(i) do
11                     foreach sequência s em tamanho de permGroup(i) do
12                         newPath ← início da sequência + sequncia(k) de
                           permGroup(i) + fim da sequência ;
13                     end
14                 end
15             end
16         end
17         adicione newPath to N ;
18     end
19     returne N ;
20 end
```

Inicialmente o Algoritmo 2 utiliza o *Princípio Fundamental da Contagem* da álgebra elementar para chegar a todas as variações possíveis em um caminho com elementos fora

de ordem (linha 5). Tal algoritmo define o número de variações multiplicando o número de permutações de cada conjunto fora de ordem. No exemplo da Figura 11, o caminho 1-2-3-4-5-6-7-8-9-10-11, com os conjuntos fora de ordem {2, 3}, {5, 6, 7} e {9,10}, possuem 24 variações. O resultado pode ser visto na Figura 12.

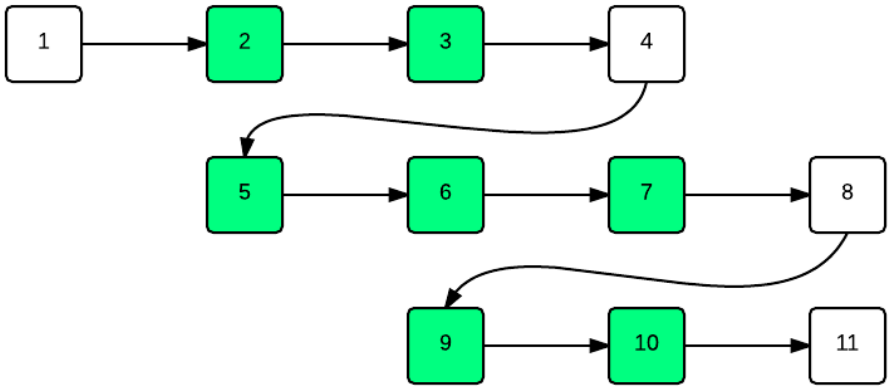


Figura 11: Caminho com múltiplos conjuntos fora de ordem

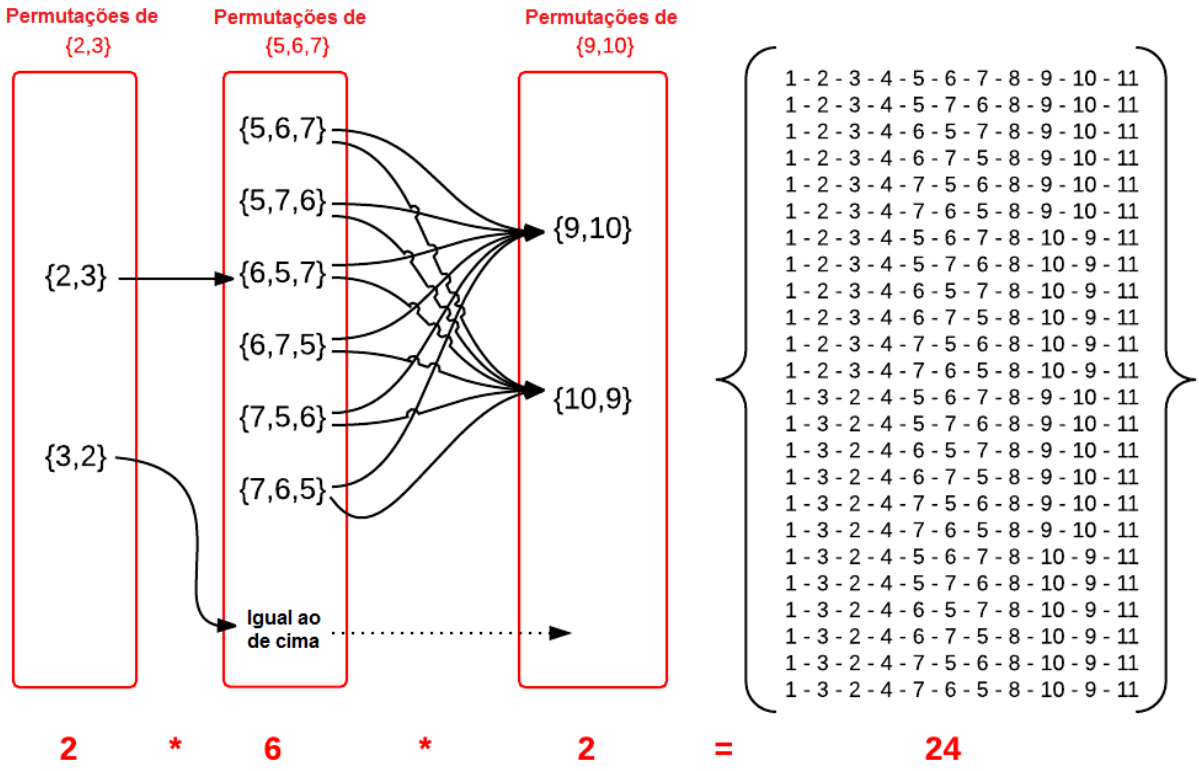


Figura 12: Princípio Fundamental da Contagem

O próximo passo do algoritmo é criar os grupos de permutação, um para cada conjunto fora de ordem (linha 8). A ideia é espalhar as permutações pelo número máximo de variações definido pelo Princípio Fundamental da Contagem, conforme mostra a Figura 12. Por exemplo, se o grupo possuir duas permutações e o máximo de variações for 24,

metade do máximo ficará com a primeira permutação, e os outros 12 elementos serão distribuídos na segunda permutação - Figura 13(A).

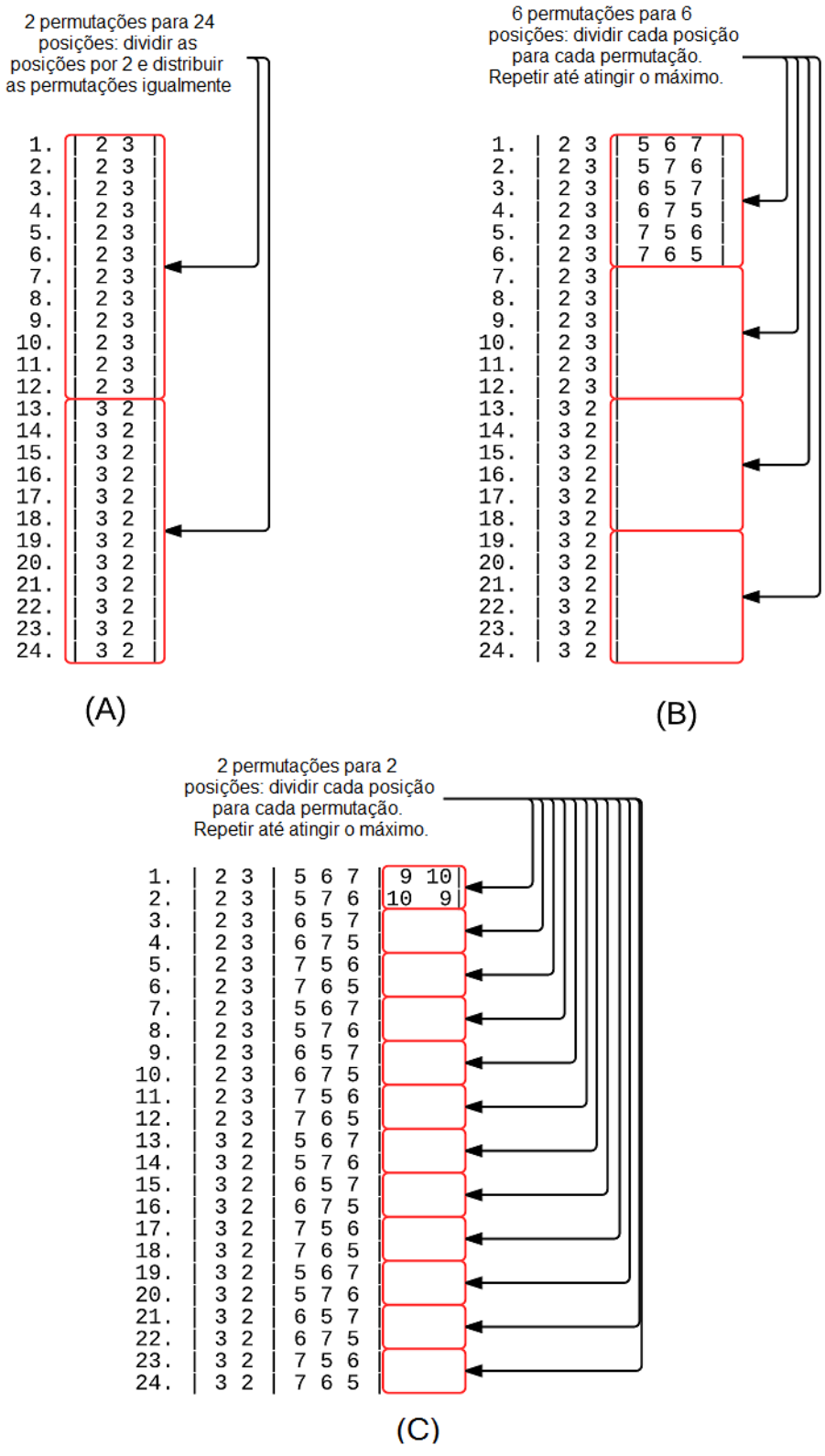


Figura 13: Grupo de permutações.

Ao processar o segundo grupo em diante, o algoritmo espalha os grupos de permutações repetidamente, até alcançar o número máximo de variações (linha 9). As Figuras 13 (B) e 13 (C) ilustram essas permutações. Após processar todos os grupos, todas as possíveis permutações são preenchidas com a correta variação, grupo por grupo.

3.2.5 Passo 5: Processar Ciclo

A ideia do Algoritmo 3 é adicionar um ciclo em todos os caminhos que já foram processados e poderiam ter um ciclo. Para isso, o algoritmo verifica a presença de eventos marcados como início de ciclo em todos os caminhos já processados (linha 5). Caso o evento esteja presente, o caminho é duplicado e o grupo de elementos que compõem o ciclo é inserido entre o evento e seu destino (linhas 11 a 15).

Algoritmo 3: Algoritmo *ProcessaCiclos*

Entrada: hash-map C com eventos de origem e destino, lista P com todos os caminhos, número de iterações de ciclos NC
Saída: lista de caminhos com ciclos CL

```
1  begin
2      foreach evento fromE em C do
3          toE ← C(fromE) ;
4          foreach sequência de eventos seq em P do
5              if fromE e toE estão em seq then
6                  cycle ← sub-lista de seq(toE, fromE) ;
7                  if cycle contém mais de 2 eventos e não processado then
8                      foreach sequência de eventos seq2 em P do
9                          if fromE está em seq2 then
10                             for i ← 0 to NC do
11                                 newSeq ← seq2(0, fromE) ;
12                                 for rep ← 0 to i do
13                                     newSeq ← newSeq + cycle ;
14                                 end
15                                 newSeq ← newSeq + seq2(fromE, end) ;
16                                 add newSeq to CL ;
17                             end
18                         end
19                     end
20                 end
21             end
22         end
23     end
24     return CL ;
25 end
```

No exemplo da Figura 14, ao processar os ciclos, verifica-se que o caminho original 1-2-3-4-5 contém um evento marcado como início de ciclo (vértice 4). Portanto, este caminho é duplicado e o ciclo {2-3-4} é inserido entre os elementos. O resultado da operação apresenta os dois caminhos: o original 1-2-3-4-5 e o caminho com ciclo 1-2-3-4-2-3-4-5.

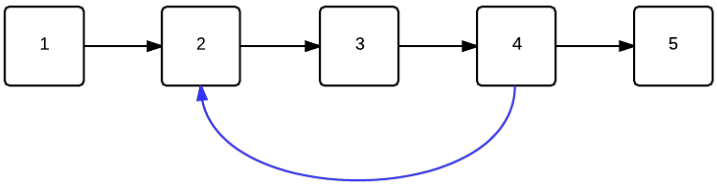


Figura 14: Grafo simples com ciclo.

Este algoritmo foi projetado para permitir ao usuário entrar com o número de iterações de ciclos que deve ser inserido nas sequências de testes (via entrada NC). O código inicia com dois operadores *for* para combinar os ciclos com os caminhos correspondentes (linhas 2 e 4). Se um ciclo está presente em um caminho específico e este ciclo possui mais de dois elementos, o caminho é duplicado e o ciclo é adicionado. Esse passo é repetido de acordo com o número de iterações desejado (linha 10).

Os ciclos somente são processados caso a sequência de ciclos possuir mais de dois elementos (linha 7). Essa restrição é justificada pelo algoritmo detalhado na Seção 3.2.1, o qual trata ciclos com apenas dois elementos. Naquele ponto, diversos ciclos foram adicionados para representar o comportamento dos elementos fora de ordem. Por este motivo, esses *mini* ciclos não são processados novamente durante a etapa corrente.

Quando um elemento marcado como início de ciclo é encontrado num caminho, o ciclo é inserido no caminho original e também em todas as variações relacionadas aos elementos fora de ordem e opcionais. O operador *for* da linha 8 é responsável por distribuir os ciclos por todos os caminhos.

3.3 Resultado da geração dos casos de teste

No final da operação dos cinco primeiros passos do UsaTasker++, todos os casos de teste são listados com as respectivas sequências de eventos. Os casos simples, sem ciclo, são listados primeiro. Em seguida, as permutações e os casos com ciclo são exibidos.

A Figura 15 apresenta a tela do UsaTasker++ com os testes gerados para a sequência de nove eventos com elementos opcionais, elementos fora de ordem e um ciclo.

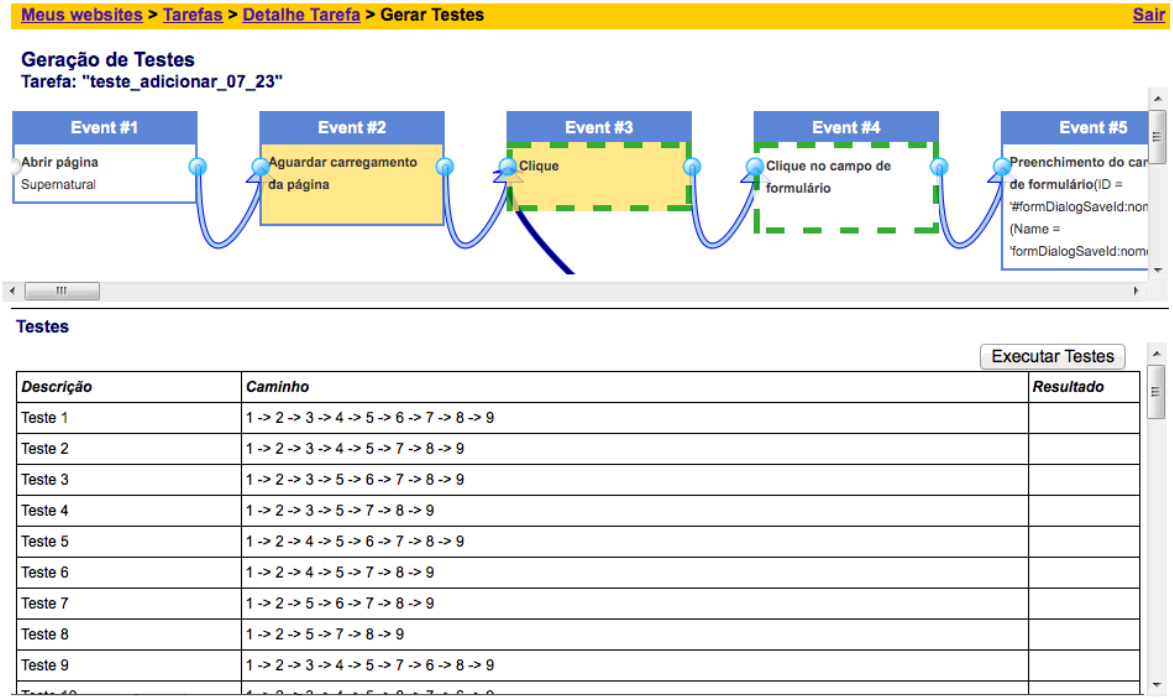


Figura 15: UsaTasker++: Casos de teste gerados

A parte superior da tela permite a consulta do grafo que representa o modelo de tarefas. Neste ponto, nenhuma manipulação pode ser feita no grafo. A categorização dos eventos como opcionais, fora de ordem ou cíclicos é realizada somente na tela anterior (relacionada a primeira versão do UsaTasker).

A parte central da figura concentra todos os casos de teste listados. A primeira coluna da lista representa o número do teste gerado. A segunda coluna representa a sequência de eventos para realização da tarefa. Os eventos são especificados pelo seu respectivo número, também relacionado em cada vértice do grafo da parte superior. Por fim, a terceira coluna apresenta o resultado da execução automática da sequência de testes, tópico da Seção 4. Tal execução pode ser disparada pelo botão "Executar Testes", disponível logo acima da terceira coluna.

As informações apresentadas na tela podem ser utilizadas de diversas maneiras. Os desenvolvedores podem selecionar aleatoriamente alguns testes para validações básicas durante a etapa de desenvolvimento. Os testadores podem utilizar o conjunto completo de testes para assegurar que todos os requisitos funcionais estão sendo verificados. Além disso, os testes podem ser executados automaticamente para fins de regressão em novas versões a serem liberadas.

Algumas abordagens, como as citadas em (BROOKS; MEMON, 2007; ZHONG; ZHANG; MEI, 2008), avaliam que a geração e execução de todos os casos de teste possíveis seja redundante e, portanto, custoso e ineficiente. A abordagem utilizada por Brooks e Memon determina que executar apenas os casos mais populares, considerando aqueles armazenados em *log*, seja suficiente para garantir a qualidade. Já Zhong et al. apresentam técnicas de heurísticas e algoritmos genéticos que determinam feitos semelhantes. Dentre as abordagens apresentadas nesses estudos, nenhuma foi capaz de garantir que todos os requisitos funcionais do sistema são verificados.

O maior risco em explorar sistematicamente a interface gráfica de uma aplicação Web é justamente a explosão combinatória do número de sequências de ações a serem executadas e o número de elementos HTML cujo estado precisa ser verificado. Uma abordagem que considera todos os testes possíveis pode ser demorada e até mesmo inviável. Este problema ainda é muito presente nos sistemas Web industriais, onde o número de elementos das páginas Web podem chegar a centenas (MCMASTER; YUAN, 2012).

Na metodologia apresentada neste trabalho e implementada pelo UsaTasker++, a facilidade com que as tarefas são gravadas e os testes são gerados, permite ao operador executar experimentos até que o número de testes gerados seja satisfatório. Para evitar as explosões combinatórias, basta determinar uma sequência de eventos mais simples para a tarefa, caso seja possível.

Por fim, em futuras pesquisas, pretende-se desenvolver passos para geração de um número reduzido de testes capazes de validar os requisitos funcionais do sistema.

3.4 Considerações finais

Neste capítulo, foram apresentadas as oportunidades vislumbradas durante as pesquisas da ferramenta de usabilidade. Foi possível perceber que, os artefatos criados durante a verificação da usabilidade também poderiam ser utilizados para análise funcional dos sistemas Web.

A primeira pesquisa resultou no desenvolvimento da ferramenta chamada de USABILICS, a qual compara as sequências de eventos armazenadas em *log* com o modelo de tarefas definido. O objetivo da análise feita pela ferramenta é determinar um índice de usabilidade da página Web sendo avaliada.

Apesar das pesquisas estarem relacionadas com testes de usabilidade, o modelo de

tarefas e o respectivo grafo gerado permitem o processamento de todos os caminhos realizáveis, possibilitando a verificação funcional do sistema. Portanto, este trabalho teve como foco a implementação do UsaTasker++, um complemento para o USABILICS capaz de realizar tal verificação.

A metodologia desenvolvida consiste de 5 passos que, ao final, produzem as variações das sequências de eventos que representam a tarefa. Dentre as técnicas utilizadas, destacam-se o Princípio Fundamental da Contagem, Permutação e Algoritmo de Busca em Profundidade.

Além da geração dos casos de teste com base em apenas uma tarefa, a ferramenta também possibilita a execução automática dos cenários criados. A economia de tempo de geração e execução dos testes permite ao desenvolvedor explorar ainda mais a verificação do sistema, otimizando a fase de testes e aumentando a qualidade do *software*.

4 Execução Automática de Testes

Um sistema real pode conter milhares de cenários de teste (THUMMALAPENTA *et al.*, 2012). Considerando que o sistema precisa evoluir rapidamente e também precisa mitigar os erros e falhas de segurança o quanto antes, é inviável passar meses verificando mudanças e analisando se isso causou algum efeito colateral. Para otimizar a verificação do sistema e exercitar comportamentos esperados (teste de regressão) é indispensável utilizar ferramentas de automação de testes. A automação de testes é considerada crucial para o sucesso de aplicações Web grandes, pois economiza muito tempo na execução dos testes e ajuda na entrega com menos defeitos (BERNER; WEBER; KELLER, 2005).

Este capítulo aborda os principais conceitos e ferramentas relacionados a automação de testes. A Seção 4.1 define um teste automatizado, enquanto a Seção 4.2 apresenta a automação no UsaTasker++. A Seção 4.3 relaciona outras ferramentas disponíveis na literatura e as principais dificuldades ainda presentes. Por fim, a Seção 4.4 destaca as técnicas utilizadas pelo UsaTasker++ para abordar os desafios da automação.

4.1 Testes Automatizados

Segundo Thummalapenta et al. (THUMMALAPENTA *et al.*, 2013), um caso de teste automatizado é um *script* que consiste de uma sequência de ações na interface gráfica da aplicação juntamente com dados relevantes. A partir da configuração das pré-condições, os testes são executados para então, comparar o resultado esperado com o real. Por fim, um relatório de teste é exibido com a indicação de sucesso ou falha.

O foco da automação é no comportamento observável das entradas e saídas, ao invés da estrutura ou estado interno do sistema ou aspectos temporais (HAUPTMANN; JUNKER, 2011). Os *scripts* são compostos de comandos para simular entradas no sistema, além de verificações para comparar as saídas.

A maioria das ferramentas de automação auxiliam no processo de gravação das ações,

na criação do *script* de teste correspondente, e na execução do *script* em um navegador. Tais ferramentas são denominadas *Capture and Replay* (gravar e repetir).

As vantagens de utilizar ferramentas *Capture and Replay* são evidentes na economia de tempo para codificar manualmente os *scripts* de teste e na simplicidade da codificação. Outro benefício é a possibilidade de executar os casos de teste quantas vezes forem necessárias de maneira simples e ágil (NEDYALKOVA; BERNARDINO, 2013).

Apesar disso, as ferramentas *Capture and Replay* também possuem desvantagens. Nedyalkova e Bernardino afirmam que, eventualmente os *scripts* gerados são ineficientes e precisam de intervenções manuais. Outro problema é que, em testes de regressão, qualquer mudança na interface gráfica do sistema pode causar falha nos testes e interromper a execução. No contexto geral, as ferramentas *Capture and Replay* são muito utilizadas no mercado e são consideradas cruciais para o sucesso de aplicações Web grandes (BERNER; WEBER; KELLER, 2005).

A Tabela 3 ilustra as ações de um caso de teste para um sistema de loja de comércio eletrônico. A página Web exibe promoções para *smartphones* do tipo Android e o objetivo do teste é verificar se a operação "*Comprar Smartphone*" funciona de maneira apropriada.

Tabela 3: Sequência de ações do caso de teste *Comprar Smartphone*

Passo	**Ação**	**Tipo**	**Nome**	**Dado**
1	entrar	caixa de texto	Buscar Produto	"smartphone"
2	clicar	botão	Buscar	
3	selecionar	link	produto 1	"Android"
4	exibir	texto	promoção	"desconto 10%"

A primeira ação do caso de teste é para inserir a palavra *"smartphone"* no campo de busca (passo 1). Ao inserir o texto desejado, pressiona-se o botão buscar (passo 2). Logo em seguida, os resultados são apresentados e a opção *"Android"* é selecionada (passo 3). Por fim, o sistema verifica se o texto *"desconto 10%"* é apresentado (passo 4).

O *script* de teste correspondente à sequência de ações é apresentado no Algoritmo 4. Os passos da Tabela 3 são mapeados no *script* a partir da linha 3. O objetivo do teste é verificar se a promoção do passo 4 é exibida. A linha 7 realiza tal verificação e, caso o texto não seja encontrado, o *script* é abortado e o teste falha. Caso contrário, o teste finaliza com sucesso.

Algoritmo 4: *Script* de teste para *Comprar Smartphone*

Entrada: *link* da loja de comércio eletrônico
Saída: Sucesso ou Falha na execução do teste automatizado

```
1  begin
2  |  Abrir link ;
3  |  Inserir "smartphone" em caixa de texto Buscar Produto ;
4  |  Pressionar botão Buscar ;
5  |  Selecionar opção "Android";
6  |  resultado = texto apresentado ;
7  |  if resultado = "desconto 10%" then
8  |  |  sucesso;
9  |  end
10 |  else
11 |  |  falha;
12 |  end
13 end
```

4.2 Automação de Testes no UsaTasker++

4.2.1 Passo 6: Automação

Um dos objetivos deste trabalho é permitir a execução automática dos casos de teste gerados pelo UsaTasker++. Para realizar tal objetivo, desenvolveu-se a abordagem *Capture and Replay* na ferramenta. Conforme o sexto passo da Figura 6 (Seção 3.2), após processar todos os caminhos do grafo, o UsaTasker++ mapeia os casos de teste em *scripts* de testes automatizados, possibilitando a execução e indicação dos cenários com erro.

O exemplo ilustrado na Figura 16 apresenta uma sequência de ações para comprar um produto em um sistema de comparação de preços na Web. Na referência (1) da figura, a página é aberta no navegador. Em seguida, o operador seleciona o campo de busca (2), insere o nome do produto (3) e pressiona o botão para pesquisar (4). O resultado é apresentado em (5) e, após optar por um produto específico (6), os detalhes são exibidos em (7). Em (8) o operador desliza a barra de rolagem até encontrar a opção de compra, a qual é selecionada em (9). Por fim, os detalhes para o pagamento começam a ser tratados em (10).

Ao executar as dez ações diretamente no navegador, o UsaTasker++ cria o grafo da Figura 17. Cada vértice do grafo processado pela ferramenta corresponde a uma ação executada.

Após a criação do grafo básico, a ferramenta permite configurar as ações em *opcional*,

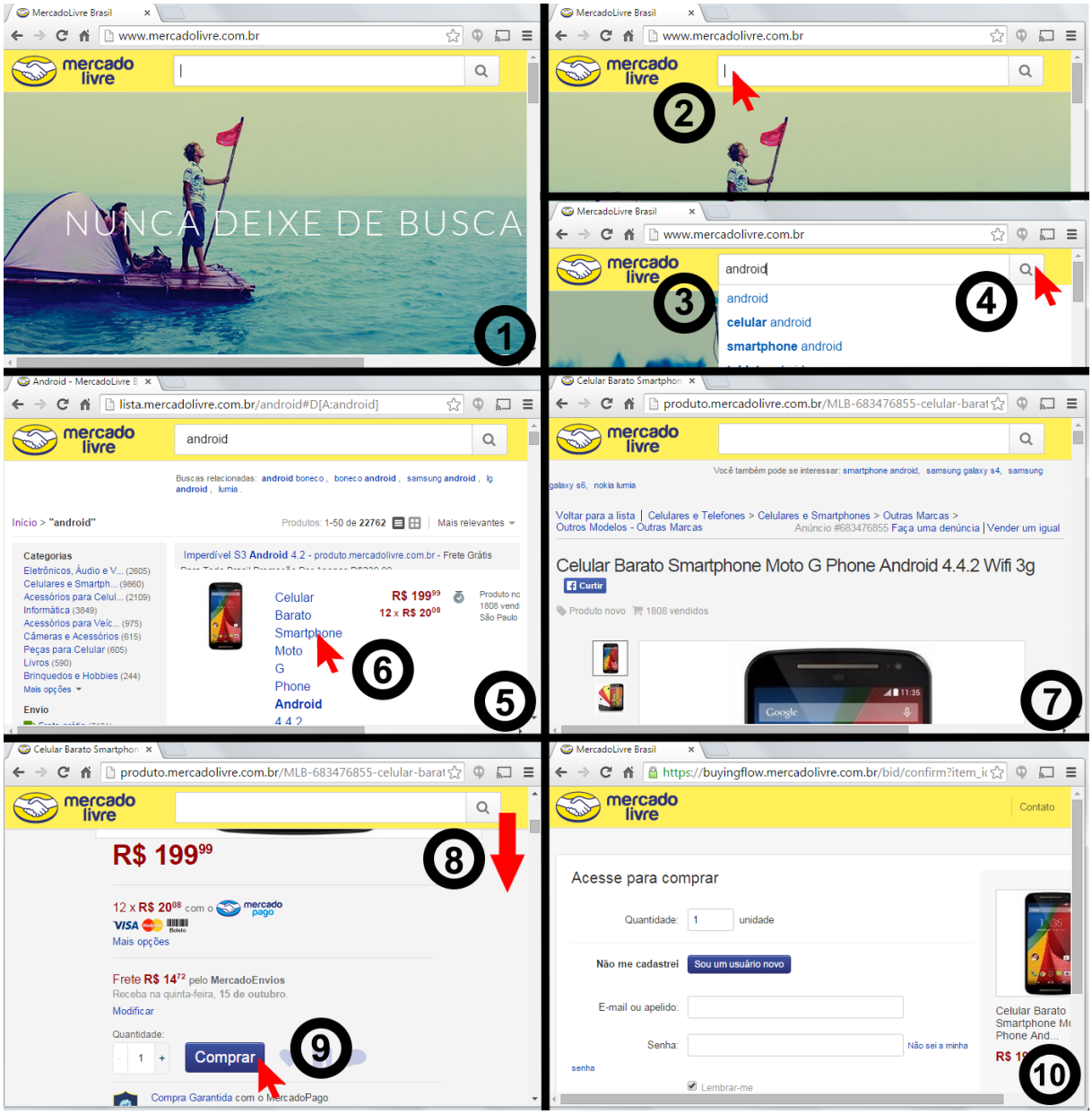


Figura 16: Sequência de eventos para a tarefa *Comprar celular*

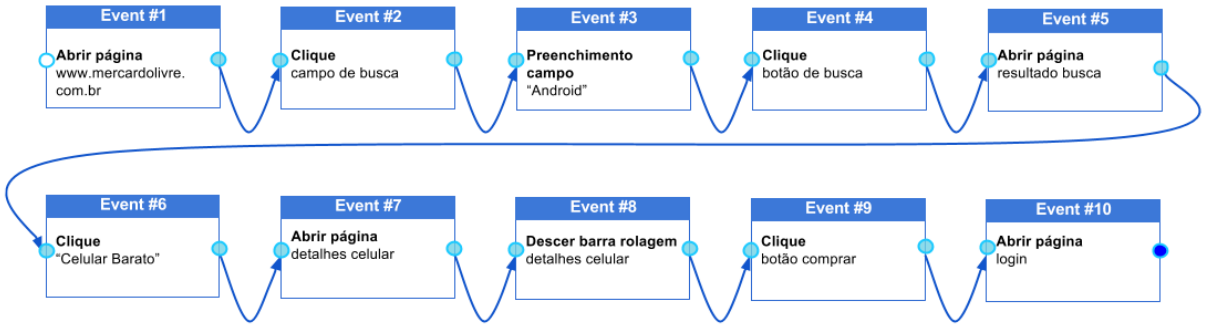


Figura 17: Grafo correspondente a sequência de eventos da tarefa *Comprar celular*

fora de ordem e *repetido*. No exemplo da Figura 18, o evento 8 foi marcado como opcional.

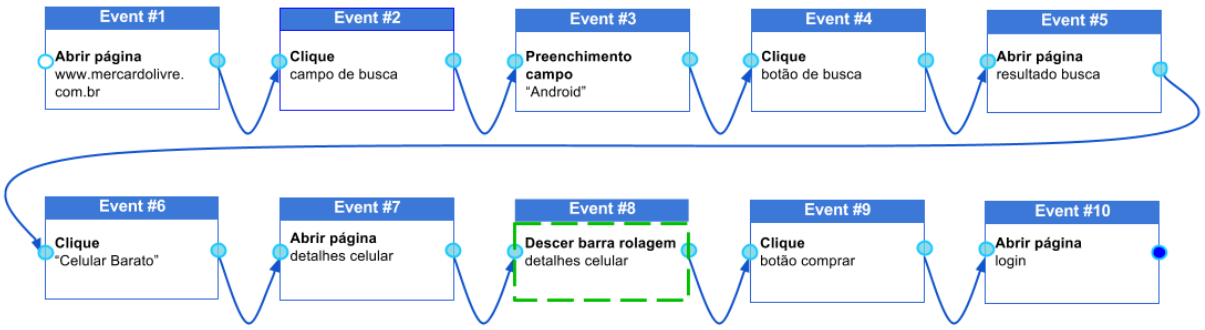


Figura 18: Grafo manipulado para a sequência de eventos da tarefa *Comprar celular*

Em seguida o UsaTasker++ gera os casos de teste correspondentes. Como apenas o evento 8 foi marcado como opcional, existem somente dois cenários possíveis: um considerando o evento 8 e outro sem o evento. A Figura 19 ilustra a tela do UsaTasker++ com os casos de teste.

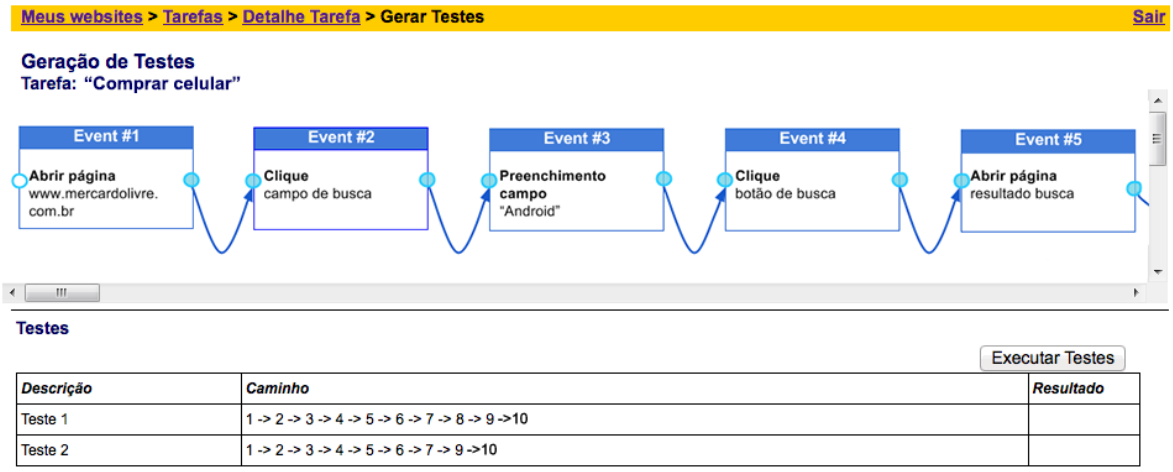


Figura 19: UsaTasker++: casos de teste gerados para a tarefa *Comprar celular*

Para montar os *scripts* de teste, cada cenário criado durante a geração dos casos de teste é convertido em uma sequência de ações a serem executadas pelo UsaTasker++ no navegador. Ao final de cada execução, o resultado é apresentado em frente ao respectivo caso de teste (Figura 20). Neste exemplo, o primeiro teste foi executado com sucesso, porém, o segundo falhou.

Com as informações apresentadas na ferramenta, o analista de teste sabe exatamente quais cenários são válidos e quais apresentam problemas. A partir desse momento, é possível investigar os cenários para encontrar os motivos do erro. Tal análise não é feita no UsaTasker++, e sim no sistema Web sendo avaliado.

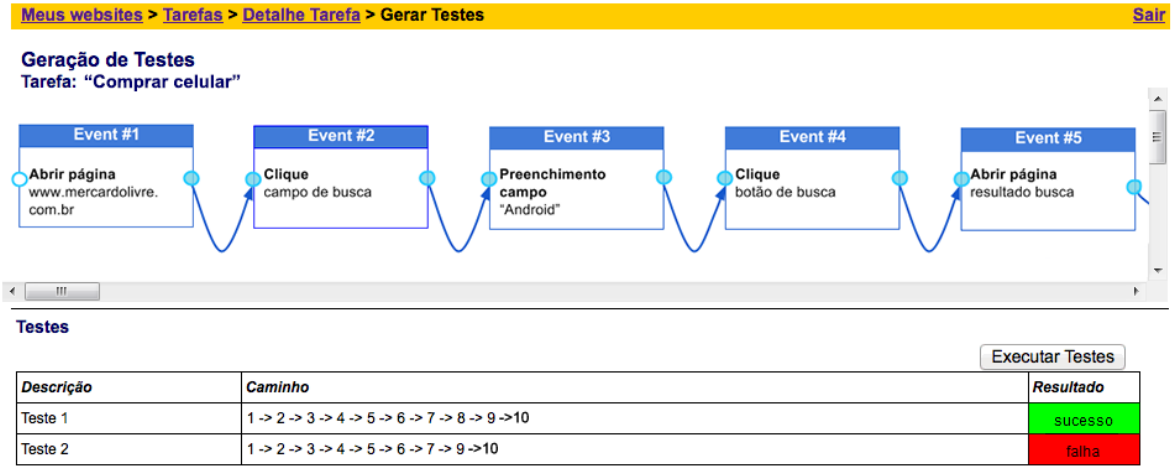


Figura 20: UsaTasker++: resultado da execução dos testes automatizados

Para implementar a automação no UsaTasker++, os recursos da interface de programação Selenium WebDriver foram adotados. Por se tratar de uma biblioteca amplamente difundida e adotada pela indústria, além de possuir código aberto, optou-se por utilizar tal API para executar os casos de teste gerados neste estudo.

4.2.2 Selenium

A API Selenium (SELENIUMHQ, 2015) é uma das bibliotecas mais populares para automação de testes de sistemas Web, além de estar empregada em diversos projetos da indústria (COLLINS; LUCENA JR., 2012). Dentre as facilidades providas, destacam-se o Selenium IDE[1], utilizado para gravar e reproduzir eventos no navegador, e o Selenium WebDriver[2], uma interface de programação para desenvolvimento de *scripts* de teste.

Conforme Leotta et al. (LEOTTA *et al.*, 2013), o Selenium WebDriver provê uma interface de programação mais compreensiva, facilitando o controle do navegador. Os casos de teste são implementados manualmente em uma linguagem de programação integrando comandos nativos do Selenium WebDriver com declarações do *JUnit*[3].

As configurações dos testes podem ser feitas, por exemplo, em código Java ao invés de arquivos de configuração complexos, diminuindo significativamente o tempo de preparação. A possibilidade de integrar o código das aplicações Web com a interface de programação facilita o desenvolvimento dos testes. Além disso, a interface de progra-

[1] http://docs.seleniumhq.org/projects/ide/
[2] http://docs.seleniumhq.org/projects/webdriver/
[3] Ferramenta para teste de unidade - http://junit.org/

mação é estável, amplamente utilizada e em constante evolução (MCMASTER; YUAN, 2012).

O Selenium WebDriver também oferece diversas maneiras para localizar um elemento da interface gráfica da página Web. A mais eficiente, de acordo com os desenvolvedores do Selenium[4], é buscando pelo identificador do elemento. McMaster e Yuan afirmam que a utilização do atributo de identificação dos elementos HTML é uma norma industrial e permite a criação de testes mais robustos se comparados a técnicas como expressoẽs XPath[5] e DOM. Dessa maneira, as funcionalidades oferecidas em uma página Web tornam-se "serviços" disponíveis pelos objetos identificáveis da página correspondente e podem ser facilmente chamados por qualquer caso de teste.


```
1  <form name="loginform" action="homepage.asp" method="post">
2   Username: <input type="text" id="UID" name="username"><br>
3   Password: <input type="text" id="PW" name="password"><br>
4   <a href="javascript:loginform.submit()" id="login">Login</a>
5  </form>
```

Figura 21: Página de login e a definição dos identificadores *UID* e *PW*

Considerando o exemplo da Figura 21 extraído de (LEOTTA *et al.*, 2013), os campos de usuário e senha são identificados no código pelos nomes *UID* (linha 2) e *PW* (linha 3), respectivamente. A interface do Selenium WebDriver permite encontrar tais elementos dentro do escopo da página Web e executar as operações desejadas.

A Figura 22 apresenta o comando *findElement* da interface de programação, utilizado para localizar os campos definidos no exemplo anterior. Com as referências localizadas, os campos de usuário e senha são preenchidos com valores enviados por parâmetro (linhas 2 e 3) e, em seguida, o botão de *Login* é acionado para efetuar a operação do clique (linha 4).

Após executar as ações, o código deve assegurar que a página encontra-se no estado desejado. O Selenium WebDriver permite verificar se o sistema foi realmente acessado pelo usuário e se a página seguiu a sequência correta de execução. Um exemplo dessa verificação pode ser vista no código da Figura 23. O comando *assertEquals* (linha 2)

[4]http://docs.seleniumhq.org/docs/03_webdriver.jsp
[5]http://www.w3.org/TR/xpath/

```
1 public HomePage login(String UID, String PW) {
2  driver.findElement(By.id("UID")).sendKeys(UID);
3  driver.findElement(By.id("PW")).sendKeys(PW);
4  driver.findElement(By.id("login")).click();
5  return new HomePage(driver);
6 }
```

Figura 22: Código de testes da página de login

efetua uma análise para determinar se o usuário que acessou o sistema é realmente o "John.Doe". Caso positivo, a verificação é bem sucedida. Caso contrário, o teste falha.

```
1 HomePage HP = LP.login("John.Doe","123456");
2 assertEquals("John.Doe", HP.getUsername());
```

Figura 23: Código de verificação da página de login

4.2.3 Selenium no UsaTasker++

O UsaTasker++ separa os motores de geração de casos de teste do sistema de automação. Conforme o diagrama de pacotes exibido na Figura 24, o USABILICS engloba, além do UsaTasker, os pacotes do UsaTasker++. Já o UsaTasker++ é composto pelo pacote de geração de casos de teste (*TestCase*) e pelo pacote de teste automatizado (*AutoTest*). As interfaces do Selenium WebDriver estão encapsuladas de tal forma no UsaTasker++, que elas podem ser substituídas futuramente com mínimos impactos para os usuários.

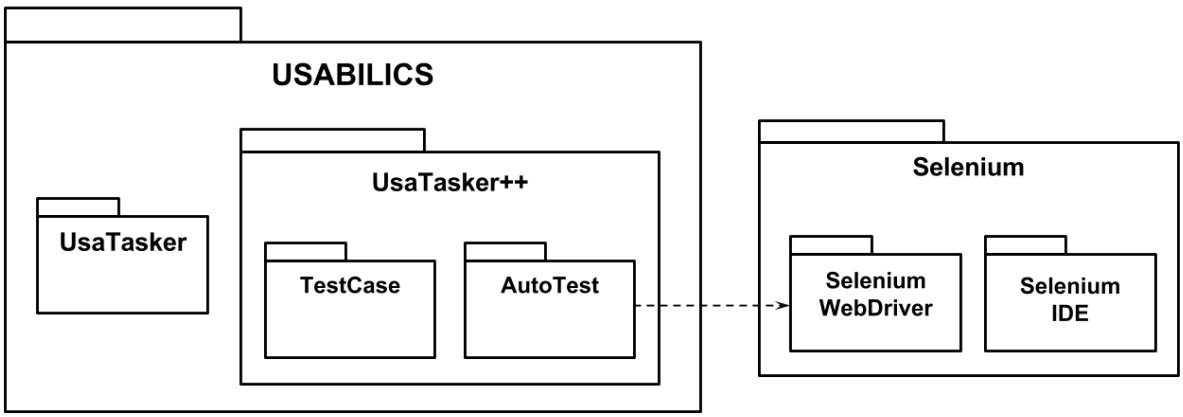


Figura 24: Diagrama de pacotes do USABILICS e o relacionamento com o Selenium

O controlador do sistema de automação mapeia cada evento presente no grafo manipulado a um comando executável no Selenium WebDriver. No exemplo da Figura 25, os eventos 1 e 3 - *click on the link* (selecionar o link) são convertidos para o comando *click()*. Da mesma maneira, o evento 2 - *open the page* (abrir a página) corresponde ao comando

getURL(). Neste exemplo, um *link* é selecionado, a página Web carregada, e então, outro *link* é selecionado.

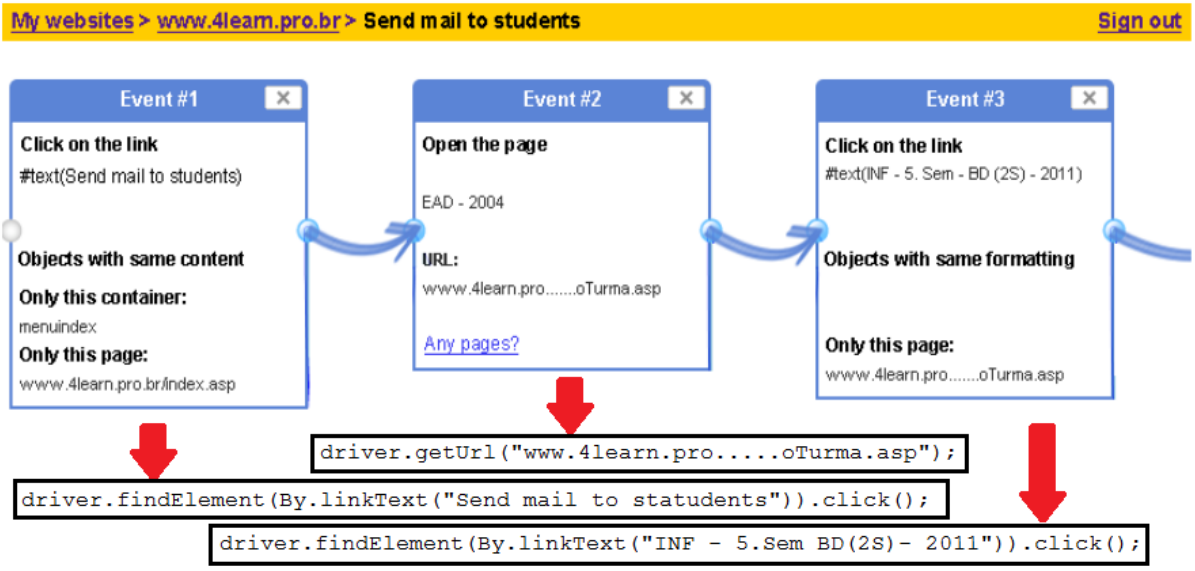
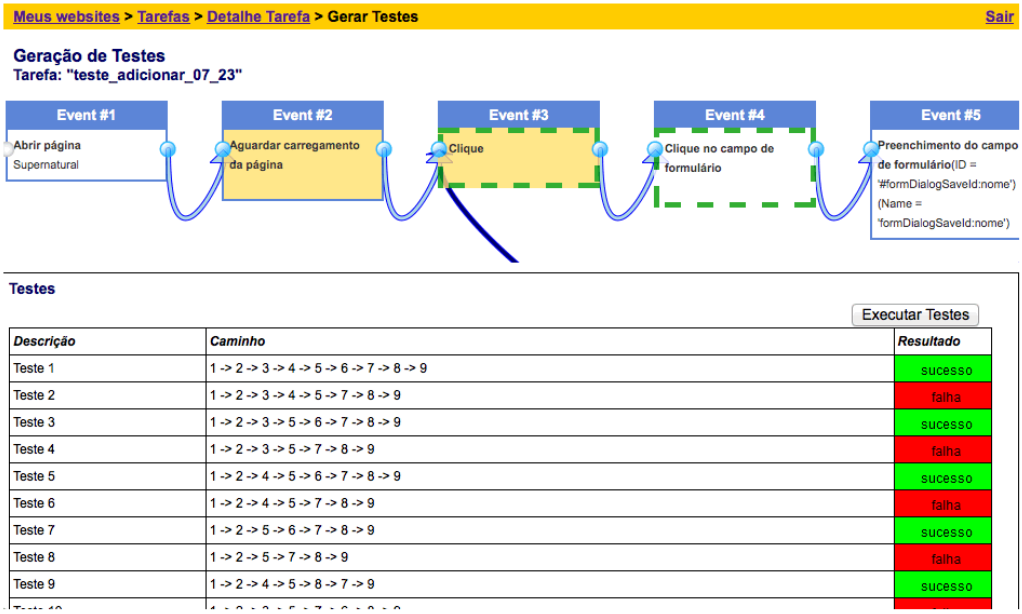


Figura 25: Mapeamento Evento - Comando Selenium WebDriver

O resultado da execução do teste é apresentado na mesma tela de testes do UsaTasker++, logo à frente da respectiva sequência de eventos. Caso o teste seja executado sem problemas, o resultado é positivo (sucesso). Se houver alguma falha, o resultado é negativo (falha). A Figura 26 ilustra esses cenários.



Descrição	*Caminho*	*Resultado*
Teste 1	1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9	sucesso
Teste 2	1 -> 2 -> 3 -> 4 -> 5 -> 7 -> 8 -> 9	falha
Teste 3	1 -> 2 -> 3 -> 5 -> 6 -> 7 -> 8 -> 9	sucesso
Teste 4	1 -> 2 -> 3 -> 5 -> 7 -> 8 -> 9	falha
Teste 5	1 -> 2 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9	sucesso
Teste 6	1 -> 2 -> 4 -> 5 -> 7 -> 8 -> 9	falha
Teste 7	1 -> 2 -> 5 -> 6 -> 7 -> 8 -> 9	sucesso
Teste 8	1 -> 2 -> 5 -> 7 -> 8 -> 9	falha
Teste 9	1 -> 2 -> 4 -> 5 -> 8 -> 7 -> 9	sucesso

Figura 26: UsaTasker++: resultado da execução dos testes

O resultado positivo acontece quando todos os comandos, após o mapeamento dos

eventos, são executados com sucesso. Se qualquer comando falha, o Selenium lança um erro e interrompe a execução do teste corrente. Neste caso, o resultado do teste fica como falho. Um exemplo de falha é quando a ferramenta executa uma ação para clicar em um *link* que não existe ou para abrir uma página não disponível na sequência.

O Selenium WebDriver permite executar os testes diretamente no navegador instalado na máquina ou em um simulador. A primeira opção possibilita que os usuários acompanhem a execução em tempo real. Já a segunda opção roda em plano de fundo, sem transmissão visual, porém, bem mais rápida e com menos recursos.

Tendo em vista que o número de casos de teste pode ser muito grande, optou-se por utilizar o simulador no UsaTasker++. Tal recurso, chamado de *HtmlUnitDriver*[6], é baseado na implementação em Java de um simulador de navegador Web sem a interface gráfica, o qual permite a execução automática de maneira leve e rápida. A desvantagem dessa abordagem é o fato de não haver qualquer acompanhamento do usuário, não sendo possível interagir ou visualizar o andamento da execução corrente.

4.3 Trabalhos relacionados

Outros trabalhos também utilizaram recursos disponíveis no mercado para suportar suas abordagens de teste em aplicações Web. McMaster e Yuan (MCMASTER; YUAN, 2012) desenvolveram uma ferramenta chamada de *WebTestingExplorer*, capaz de criar um modelo de estados conforme o usuário navega pela aplicação. À medida que o modelo é incrementado, novos testes são gerados. Para realizar tais funções, o sistema utiliza a linguagem de programação Java em conjunto com as bibliotecas do Selenium. Apesar das semelhanças, essa abordagem difere do UsaTasker++ nos tipos de testes gerados. Por não possuir mecanismos de classificação de eventos, os testes gerados no *WebTestingExplorer* se baseiam apenas no fluxo percorrido pelo usuário.

Já o sistema gerado por Thummalapenta et al. (THUMMALAPENTA *et al.*, 2013) utiliza técnicas de varredura para criar um modelo navegacional e mapear as regras de negócio da aplicação. Em seguida, o sistema é capaz de identificar os caminhos mais interessantes e gerar casos de teste. A plataforma utilizada para geração e execução dos testes foi a IBM Rational Functional Tester[7]. Apesar da abordagem não precisar da interação humana para gerar o modelo navegacional, os testes demonstraram que a

[6]http://docs.seleniumhq.org/docs/03_webdriver.jsp#htmlunitdriver
[7]http://www03.ibm.com/software/products/pt/functional

ferramenta consegue cobrir, em média, 92% das regras de negócio.

No modelo definido por Leotta et al. (LEOTTA *et al.*, 2013), ao invés de referenciar os elementos HTML pelo seus identificadores, os autores desenvolveram diversas técnicas para reconhecimento dos objetos de interface. Com isso, os *scripts* de teste automatizados gerados não são limitados pelas páginas dinâmicas, visto que se baseiam no conteúdo, texto, formato e outras características dos elementos. Nesta abordagem, apesar da ampla geração de testes automatizados, o foco é a manutenibilidade dos mesmos. Tanto a regra de reconhecimento dos elementos HTML, quanto a geração dos *scripts* de teste são desenvolvidos em conjunto com as bibliotecas do Selenium WebDriver.

Na mesma linha, Al-Zain et al. (AL-ZAIN; ELEYAN; GARFIELD, 2012) também criaram um sistema focado na facilidade de manutenção dos casos de teste. A abordagem definida por eles utiliza *scripts* para atualizar os cenários de teste caso a aplicação sofra alterações na interface gráfica. Neste caso, o sistema utiliza os recursos da plataforma ASP.NET para realizar a gravação, reprodução e atualização dos casos de teste.

Já o foco de Barbosa et al. (BARBOSA; PAIVA; CAMPOS, 2011) é na geração de cenários de testes com base em modelos de representação. O diferencial da abordagem criada por eles é a introdução de erros comuns executados pelos usuários, gerando novos casos de teste. Além disso, os *scripts* de teste são escritos na linguagem Spec#[8] com o intuito de facilitar a utilização dos desenvolvedores, visto que possui uma notação visual.

Papadakis et al. (PAPADAKIS; MALEVRIS; KALLIA, 2010) criaram uma ferramenta para geração de dados para testes automatizados. Eles utilizaram um conjunto de técnicas para gerar variações de entradas que eram distribuídas em cenários preestabelecidos. Cada entrada diferente aplicada a cada cenário consiste em um caso de teste único. Apesar de muito eficaz, existe um grande esforço para definição e manutenção dos cenários utilizados como base.

Na mesma direção, Bauersfield et al. (BAUERSFELD; WAPPLER; WEGENER, 2011) também criaram uma ferramenta para encontrar sequências de entradas para geração de testes automatizados. No trabalho realizado por eles, o algoritmo da otimização da colônia de formigas foi utilizado em conjunto com funções de aptidão. Neste caso, diversos cenários são gerados, porém, não há relação direta com a validação funcional do sistema.

No contexto de plataforma de testes, Wang e Du (WANG; DU, 2012) foram os únicos

[8]Extensão da linguagem C#, definida em http://research.microsoft.com/enus/projects/specsharp/

encontrados na literatura a desenvolver uma ferramenta que centraliza testes funcionais e não-funcionais. Para isso, os autores utilizaram o JMeter[9] para realizar testes de desempenho, enquanto o Selenium foi utilizado para gravar as ações sendo executadas na aplicação e para gerar os *scripts* de teste. A desvantagem em relação ao UsaTasker++ é a abrangência dos testes, visto que o modelo do trabalho dos autores não relaciona o conjunto de testes a ser utilizado. Já no UsaTasker++, ao utilizar a interface da aplicação Web, é possível gerar os diversos testes automatizados, os quais podem ser utilizados tanto para verificação funcional, quanto para teste de carga, conforme discutido na Seção 5.2.

4.4 Desafios da automação

Apesar das facilidades apresentadas pelos testes automatizados, alguns aspectos podem comprometer os benefícios de sua adoção. Hauptmann e Junker (HAUPTMANN; JUNKER, 2011) indicam que a manutenção dos cenários é custosa em sistemas com mudanças recorrentes nos objetos da interface gráfica, obrigando a atualização de todos os testes que contemplam o objeto modificado. Além disso, eles sugerem que o reuso do código entre os testes seja baixo devido às diferentes entradas e variantes utilizados.

Tais dificuldades não estão presentes na abordagem utilizada pelo UsaTasker++. Como os casos de teste são gerados a partir de um único cenário executado diretamente na aplicação Web, caso haja alguma alteração nos objetos da interface gráfica, basta executar o cenário básico novamente e todos os casos de teste são recriados. Neste caso, todos as variações são gerenciadas diretamente pelo sistema, tornando desnecessária a reutilização do código de teste, já que tal código não fica disponível para o usuário.

Por outro lado, caso as variações de cenários estejam relacionadas a configurações que não são contempladas pelas categorias *opcional*, *fora de ordem* e *repetido*, diferentes cenários básicos devem ser criados no UsaTasker++. Por exemplo, se o objetivo do teste é verificar o comportamento de uma entrada numeral, deve-se criar um cenário com valores positivos, outro com valores negativos e outro com valores nulos. A desvantagem dessa abordagem é que, quanto maior o número de cenários básicos, maior o esforço para mantê-los.

De uma forma geral, McMaster e Yuan (MCMASTER; YUAN, 2012) sustentam que a automação dos testes seja o caminho mais confiável para atingir a qualidade. Caso

[9]Sistema desenvolvido pela Apache para teste de carga, definida em http://jmeter.apache.org/

os testes de sistema tenham que ser executados repetidamente, como em testes de regressão, a automação pode ser muito eficiente (HAUPTMANN; JUNKER, 2011). No contexto do UsaTasker++, a ferramenta demonstrou ser eficaz mesmo diante dos desafios apresentados.

4.5 Considerações finais

Este Capítulo abordou os principais conceitos relacionados a testes automatizados, detalhando as estruturas e o funcionamento dos *scripts* de teste. Apesar dos benefícios gerados com a utilização da automação, principalmente os relacionados à economia de tempo e redução de custo, ainda existem empecilhos relacionados ao tema. Mesmo assim, a automação de testes é o caminho mais confiável para atingir a qualidade no sistemas Web.

Diversas ferramentas foram desenvolvidas para suportar a geração e execução dos testes automatizados. Dentre os tipos mais utilizados, encontram-se as ferramentas *Capture and Replay* (gravar e repetir). O foco dessas ferramentas é utilizar a própria interface gráfica das aplicações para gerar os *scripts* de teste, e depois, executar tais *scripts* sem a intervenção humana.

Uma das bibliotecas mais utilizadas na indústria é o Selenium. A API provê uma interface robusta para interação com navegadores, além de suportar as propriedades *Capture and Replay*. Dessa forma, é possível associar o código da biblioteca com os sistemas de geração e execução de testes.

Seguindo a linha do mercado, o UsaTasker++ adotou o Selenium como biblioteca padrão de suporte aos testes automatizados. Uma camada de automação foi criada para separar a lógica de geração de casos de testes da lógica de criação dos *scripts* de teste. Por fim, todos os testes criados pela ferramenta podem ser executados automaticamente em um navegador real ou simulado, fornecendo um indicativo de sucesso ou falha no relatório de testes.

Para a validação da proposta, o Capítulo 5 apresenta os detalhes da execução dos testes automatizados, além de descrever os experimentos realizados em aplicações reais.

5 Estudos de Caso e Resultados

Dentre os desafios do cenário atual da Web, há recorrentemente um curto espaço de tempo entre a concepção e a implementação das aplicações. Neste contexto, a necessidade por mecanismos de validação mais rápidos e eficazes é evidente. Abordagens manuais e custosas tornam-se inviáveis para a fase de testes. Portanto, no intuito de prover uma maneira simples e eficiente na geração e automação de casos de teste, o UsaTasker++ foi implementado.

Para validar a abordagem proposta no UsaTasker++, uma avaliação empírica foi realizada. Duas aplicações baseadas em tecnologias renomadas, como AJAX, MongoDB e NodeJS, foram utilizadas. O objetivo deste estudo foi avaliar a eficácia do modelo na geração dos casos de teste, medir se os cenários criados pelo UsaTasker++ eram capazes de cobrir os requisitos funcionais das aplicações Web, e verificar se a execução automática dos testes poderiam identificar os possíveis erros nos sistemas.

Portanto, este capítulo apresenta os detalhes dos estudos de caso realizados e os resultados obtidos. A Seção 5.1 mostra a primeira bateria de testes realizada sobre uma aplicação Web, a qual provê financiamento com taxas de juros diferenciadas para uma marca específica de automóveis. Já a Seção 5.2 consolida os resultados do segundo sistema verificado – uma aplicação de gerenciamento de projetos ágeis. Por fim, a Seção 5.3 analisa os experimentos e relaciona as lições aprendidas.

5.1 Aplicação para Financiamento de Automóveis

Os primeiros testes foram executados na aplicação de financiamento de automóveis (Figura 27). A fase inicial do experimento envolveu definir qual a principal tarefa da aplicação e quais eram os requisitos funcionais relacionados a essa tarefa. Considerando a tarefa *"Comprar Veículo"*, os seguintes requisitos foram definidos:

I. O usuário pode enviar uma proposta de apenas um veículo por vez, porém, diversas

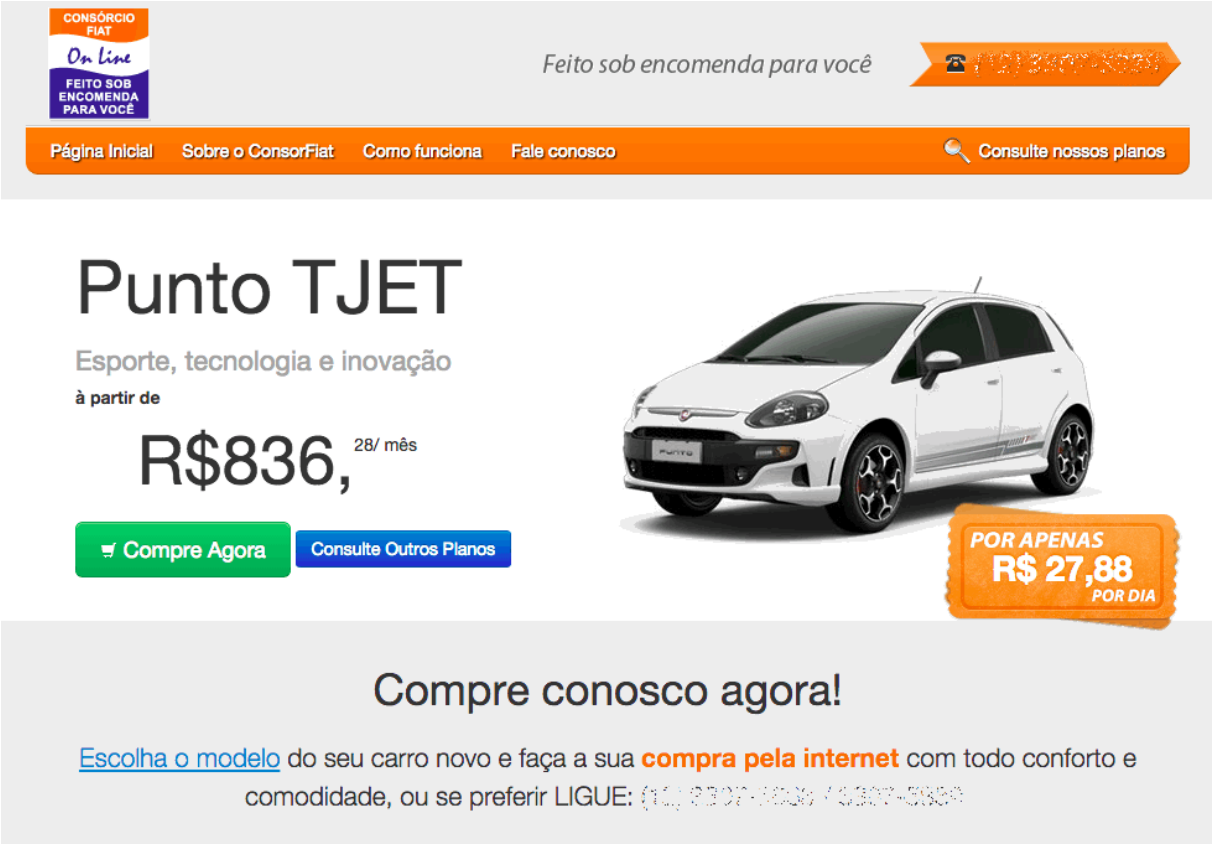


Figura 27: Aplicação para financiamento de Automóveis

propostas podem ser enviadas.

II. *Nome* e *Endereço* são campos obrigatórios, enquanto *Complemento* é opcional.

III. O usuário deve aceitar os "Termos de Uso". Caso contrário, um erro deve ser exibido.

IV. Os campos na página podem ser preenchidos em qualquer sequência.

Na segunda fase gravou-se a sequência básica de eventos que representam a tarefa. Neste momento, os eventos foram caracterizados como opcionais, fora de ordem e/ou cíclicos, com base no requisito funcional fornecido na fase anterior. O grafo da Figura 28 representa o modelo para a tarefa definida, já relacionando as configurações (*opcional, fora de ordem, repetido*) para cada tarefa.

A sequência de eventos definida nesta etapa está diretamente relacionada com o sucesso da cobertura dos casos de teste. Caso um caminho muito curto seja definido, é provável que a sequência de eventos não seja suficiente para cobrir todos os requisitos funcionais. O analista de testes deve preparar o fluxo de ações de maneira apropriada com o intuito de realizar uma verificação completa do sistema. Para fins de experimentos,

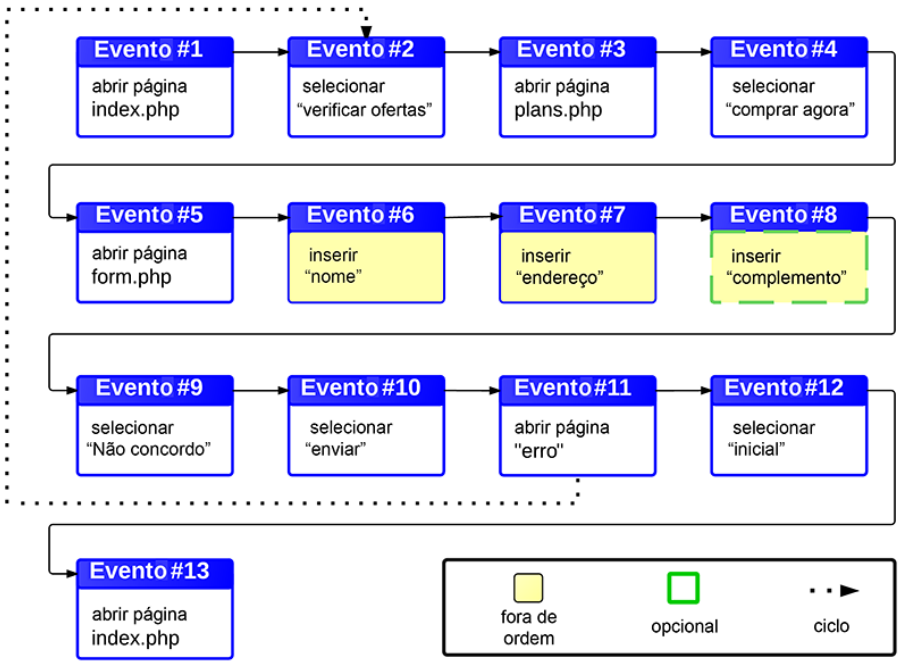


Figura 28: Tarefa *"Comprar Veículo"* - caminho reduzido

este trabalho analisou um caminho reduzido exibido na Figura 28 e depois comparou os resultados com outra abordagem mais completa.

A última fase do experimento consistiu em verificar se os casos de teste gerados foram capazes de validar todos os requisitos funcionais da aplicação, além de verificar se tais casos realmente implicam em todos os casos de teste possíveis dada a sequência básica de eventos. Para o grafo exibido na Figura 28, o UsaTasker++ gerou 72 casos de teste no total: 8 sem ciclos (chamados de *Simples*) e 64 com ciclos (chamados de *Cíclicos*).

Tabela 4: Testes gerados

Abordagem	**CI**	**FDO**	**Opc**	**Cc**	**Total**
Reduzido	1	6	2	64	72
Reduzido	3	6	2	192	200
Estendido	1	60	4	4.096	4.160
Estendido	3	60	4	12.288	12.352

CI: número de iterações dos ciclos; FDO: fora de ordem; Opc: opcional; Cc: caminhos com ciclo.

Conforme indicado na primeira linha da Tabela 4, dois casos estão relacionados com o elemento opcional (um teste com o elemento e outro teste sem o elemento), e seis casos estão relacionados com os elementos fora de ordem (derivados da permutação de 3). Cada um desses oito testes Simples possuem um ciclo diferente em sua sequência de eventos. Portanto, sessenta e quatro testes Cíclicos são originados de operações que relacionam cada ciclo aos casos de teste Simples: oito sequências de ciclos multiplicadas por oito

testes Simples.

Na segunda linha da Tabela 4 o mesmo experimento foi executado, porém, agora considerando 3 iterações de ciclo. Com isso, o número de casos de teste com ciclos foi multiplicado por 3, visto que foi criado um caso com apenas uma iteração, outro caso com duas iterações e outro caso com três. O total de testes gerados foi 200: 8 casos Simples e 192 casos Cíclicos.

Conforme as regras de permutação descritas na Seção 3.2, assim como a lógica definida para elementos opcionais e cíclicos, esse experimento obteve êxito em gerar todos os casos de teste possíveis a partir da sequência utilizada como base. No entanto, os casos de teste gerados não foram capazes de validar os requisitos funcionais I e III. Apesar do grande número de testes gerados, nenhum deles efetivamente disparou uma oferta de financiamento, visto que eles não aceitaram os "Termos de Uso". Todos os testes manipularam os elementos da interface gráfica (GUI) repetidamente, verificando o comportamento em cenários alternativos. Ficou claro que o conjunto reduzido de ações gravadas pelo Analista de Testes não passou por todas as possibilidades da GUI, deixando alguns cenários de fora do escopo.

Depois de revalidar o cenário utilizado como base, uma abordagem estendida foi determinada, conforme a Figura 29. Nessa nova tentativa, ambos os fluxos (*Não aceito* e *Aceito* os "Termos de Uso") foram gravados no mesmo caso de teste. Como resultado, o número de casos de teste aumentou para 4.160 e, ainda mais relevante, os casos de teste gerados agora foram capazes de cobrir todos os requisitos funcionais da aplicação.

Os 64 casos de teste Simples criados durante essa validação (terceira e quarta linha da Tabela 4) tiveram sucesso ao verificar os diversos fluxos originados das variações em torno do "Termo de Uso". Em seguida, cada sequência com ciclos foi distribuída em cada caso de teste Simples, gerando 4.096 casos Cíclicos (64 Simples multiplicado por 64 com ciclos). Ao configurar o número de iterações dos ciclos, o número de casos Cíclicos triplica, resultando em 12.352 cenários de teste únicos.

Conforme observado na Tabela 4, o número de casos de teste gerados é proporcional a expansividade do cenário utilizado como base. Não obstante, os resultados ressaltam a quantidade significativa de casos de teste únicos e a habilidade em gerar todas as sequências possíveis. Ainda, os testes foram capazes de estressar os requisitos funcionais do sistema.

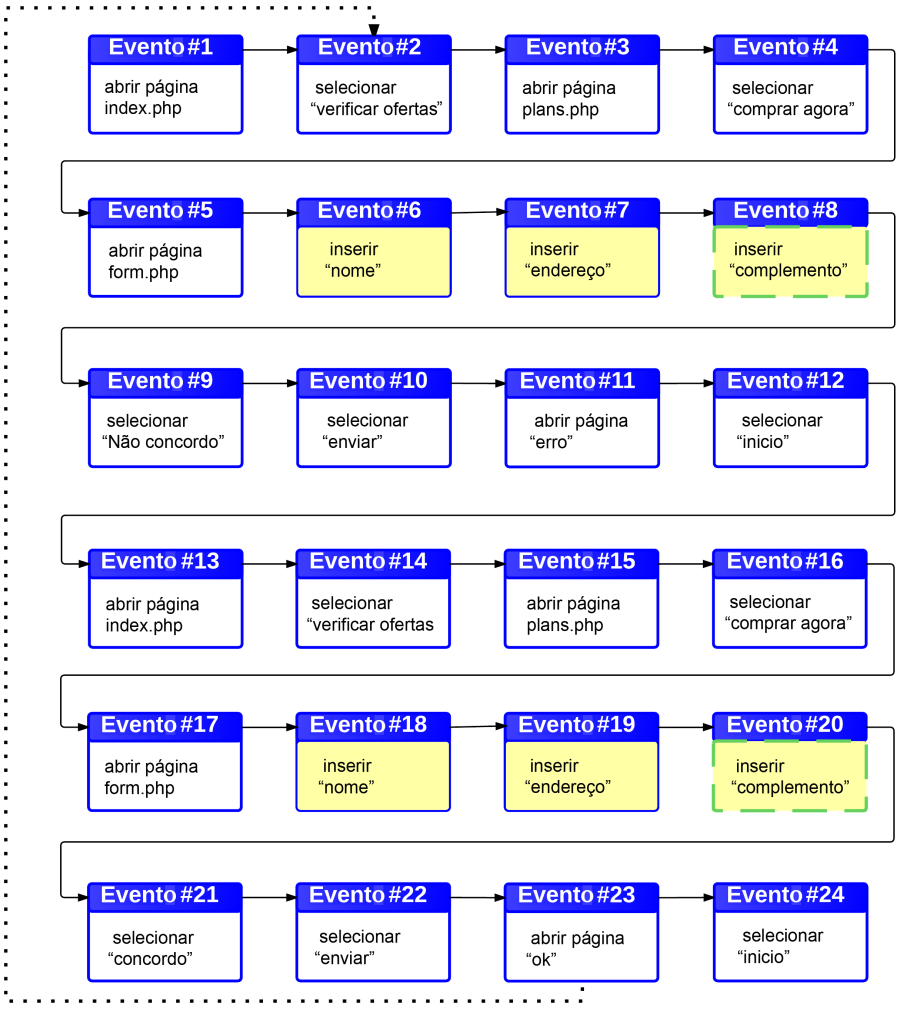


Figura 29: Tarefa *"Comprar Veículo"* - caminho estendido

Por fim, a execução do conjunto de testes desta aplicação apenas confirmou o grau de confiabilidade do sistema Web, visto que nenhum erro foi encontrado. Em futuras versões do sistema, os mesmos testes poderão ser utilizados para regressão. Caso algum teste venha a falhar, pode-se afirmar que as modificações nas novas versões inseriram um defeito. Neste caso, a análise do código defeituoso é limitada ao código alterado, simplificando os passos para a correção do erro.

5.2 Aplicação para Gerenciamento de Projetos Ágeis

A aplicação para gerenciamento de projetos ágeis, exibida na Figura 30, é utilizada para controlar o progresso das tarefas que compõe um projeto. As principais funções utilizadas no sistema são a criação de um projeto e o cadastro das respectivas atividades a serem executadas durante o projeto. Os requisitos funcionais envolvidos são listados a

Figura 30: Aplicação de gerenciamento de projetos ágeis

seguir:

I. O usuário pode criar um Projeto, devendo assinalar o seu nome e as datas de início e fim. Opcionalmente pode-se determinar a sua descrição.

II. Cada Projeto pode conter os Módulos que necessitar, devendo assinalar o seu nome.

III. Cada Módulo pode conter os Casos de Uso que necessitar, devendo assinalar o seu nome e, opcionalmente, a sua descrição.

IV. Cada Caso de Uso pode conter as Atividades que necessitar, devendo assinalar o seu nome e, opcionalmente, a sua descrição.

V. Cada Atividade deverá ter um estado previamente configurado para o projeto, como Aberto, Em andamento ou Concluído.

VI. As atividades podem ter subatividades, assinaladas com o seu nome e respectivo ponto de complexidade.

VII. Os pontos de complexidade relativos a uma atividade ou subatividade podem ser consumidos à medida que a subatividade é finalizada.

Uma sequência básica com 33 eventos foi gravada, percorrendo o fluxo desde a criação do Projeto ao consumo dos pontos de complexidade. A Figura 31 representa tal sequência e os eventos classificados conforme as suas características. Neste caso, 46.208 testes foram

gerados considerando apenas uma iteração de ciclo, e 133.760 teste com três iterações de ciclo (Tabela 5).

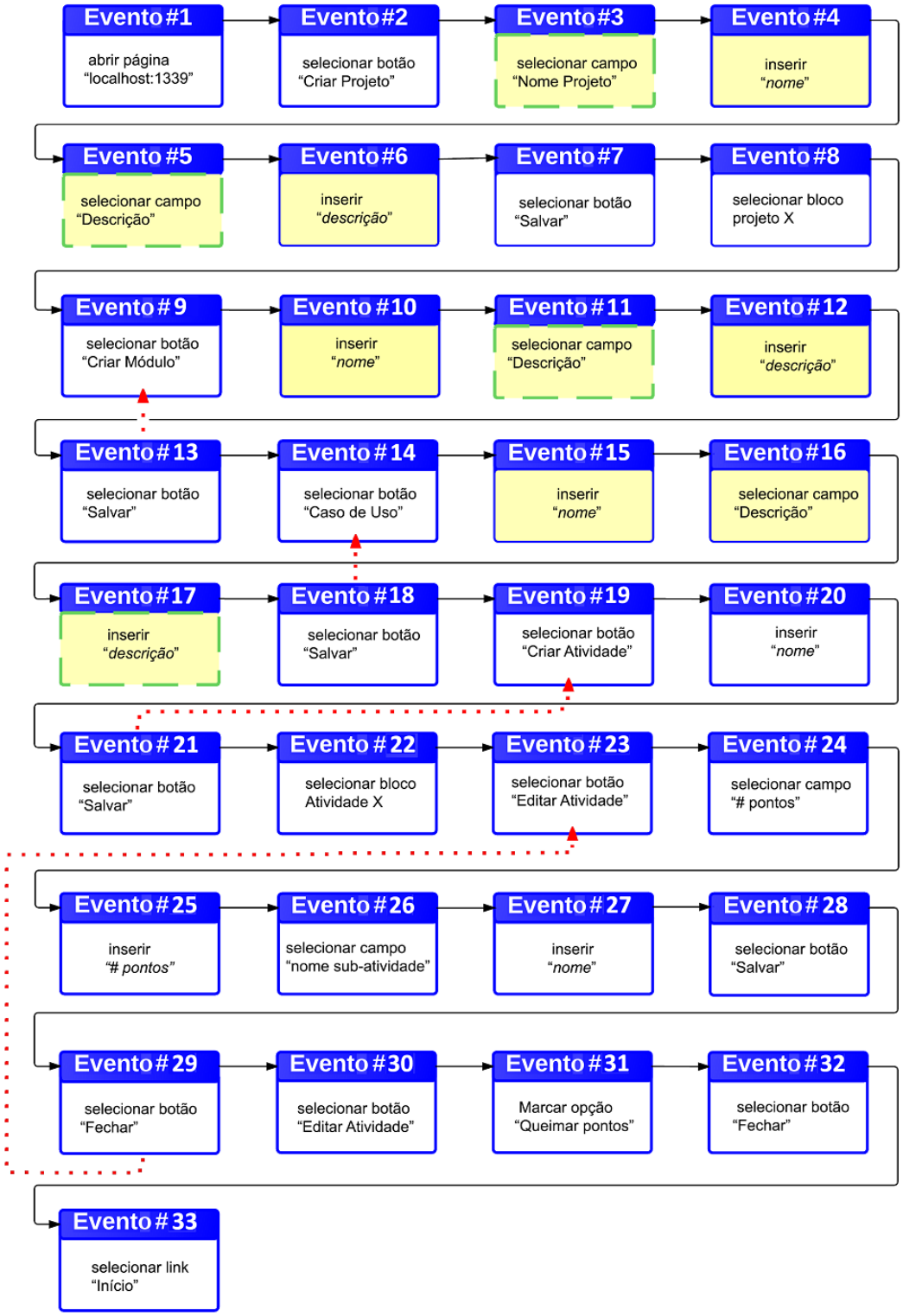


Figura 31: Grafo com eventos relacionados à tarefa de execução de Projetos.

Tabela 5: Testes gerados da primeira sequência definida

Abordagem	CI	FDO	Opc	Cc	Total
Completo 1	1	1.024	16	45.168	46.208
Completo 1	3	1.024	16	132.720	133.760

CI: número de iterações de ciclo; FDO: fora de ordem; Opc: opcional; Cc: caminhos com ciclo.

Alguns problemas aconteceram durante a execução dos testes automatizados. O *software* de gerenciamento de projetos ágeis utiliza uma técnica de criação de identificadores de elementos HTML de interface (elementos DOM[1]) de forma dinâmica. Por se tratar de informações criadas durante o tempo de execução, seus elementos HTML também possuem identificadores versáteis gerados a partir da biblioteca disponível do MongoDB[2], cujo tamanho corresponde a uma cadeia de caracteres de 12 bytes (4 bytes representando os segundos do Sistema Operacional, 3 bytes representando o identificador da máquina, 2 bytes representando o identificador do processador e 3 bytes representando um contador). Dessa forma, é possível garantir que, caso diversos usuários utilizem o sistema em paralelo, identificadores únicos serão criados para os elementos HTML.

Conforme discutido na Sessão 4.2.2, o Selenium utiliza o identificador do elemento para manipulá-lo diretamente no navegador. Se em cada caso de teste um projeto ágil e suas respectivas atividades forem criadas, cada elemento HTML terá um identificador diferente. Neste caso, não é possível utilizar o identificador armazenado durante a fase de gravação, visto que o mesmo referencia um elemento único e específico. Caso isso ocorresse, o Selenium estaria manipulando os elementos criados durante a gravação e não aqueles criados no teste corrente (e com novos identificadores).

Para solucionar tal problema, o mapeamento realizado na Sessão 4.2 teve que ser modificado para considerar os identificadores dinâmicos. A lógica de negócio foi embutida no mapeamento para que o sistema tenha ciência de quais elementos estão sendo manipulados naquele momento. Posteriormente, durante a execução dos testes automatizados, o sistema deve considerar o mesmo elemento, porém, com o identificador relativo a execução atual.

A Tabela 6 demonstra como o mapeamento é realizado para os casos estáticos e dinâmicos. Na primeira linha, o identificador "*nome*" foi utilizado diretamente na execução do evento (dentro da função *By.id*). Já na segunda linha, o sistema identifica que existe um identificador que precisa ser substituído (i.e. "*_553da0f*"), pois ele reconhece o caractere

[1]http://www.w3.org/DOM/
[2]http://docs.mongodb.org/manual/reference/objectid/

Tabela 6: Mapeamento de eventos

	Gravado				Executado
	Ev.	**Elem.**	**Id.**	**Valor**	**Mapeamento**
Estát.	click	INPUT	*nome*	Paulo	`driver.findElement(By.id("nome")).sendKeys("Paulo");`
Dinâm.	click	INPUT	*nome_553da0f*	Paulo	`driver.findElement(By.id("nome_63ab21")).sendKeys("Paulo");`

Ev.: eventos; Elem.: elementos; Id.: identificador.

subtraço ("_"). Neste caso, em cada teste executado, um novo identificador é gerado e embutido no comando (i.e. "*_63ab21*").

Devido à limitação de tempo para este trabalho, a lógica para o mapeamento é codificada juntamente com as outras funções do UsaTasker++. Futuramente, pretende-se expandir essa funcionalidade para um arquivo de configuração.

Após a etapa de mapeamento, foi possível continuar com a atividade de execução automática dos testes. Entretanto, todos os testes falharam. Percebeu-se que o requisito funcional V não estava sendo considerado, já que nenhum estado – dentre *Aberto, Em andamento* ou *Concluído* – foi definido para as atividades do projeto. Neste caso, o UsaTasker++ obteve êxito ao evidenciar que as ações obrigatórias não estavam sendo executadas, corrompendo um dos requisitos funcionais da aplicação. Portanto, uma nova sequência era necessária.

A segunda sequência básica de eventos gravados também contempla desde a criação do Projeto ao consumo dos pontos de complexidade. Porém, nesta nova tentativa, a criação dos estados foi considerada. Os eventos 9, 10 e 11 da Figura 32 representam tal ação.

Tabela 7: Testes gerados da segunda sequência

Abordagem	**CI**	**FDO**	**Opc**	**Cc**	**Total**
Completo 2	1	1.024	16	47.600	48.640
Completo 2	3	1.024	16	140.016	141.056

CI: número de iterações dos ciclos; FDO: fora de ordem; Opc: opcional; Cc: caminhos com ciclo.

A Tabela 7 apresenta os resultados da geração dos casos de teste para a sequência definida. Neste caso, há vários grupos com elementos fora de ordem no decorrer da sequência, o que proporciona um aumento exponencial em relação ao número de casos gerados. Por exemplo, os três grupos de elementos fora de ordem resultam em 1.024 sequências distintas originadas da multiplicação de cada grupo: 16 sequências do primeiro grupo (eventos 3, 4, 5 e 6), 8 sequências do segundo grupo (eventos 13, 14 e 15) e 8 do terceiro grupo (eventos 18, 19 e 20). Além disso, cada ciclo é distribuído por cada caminho Simples e as respectivas variações, totalizando 48.640 casos de teste com uma iteração de

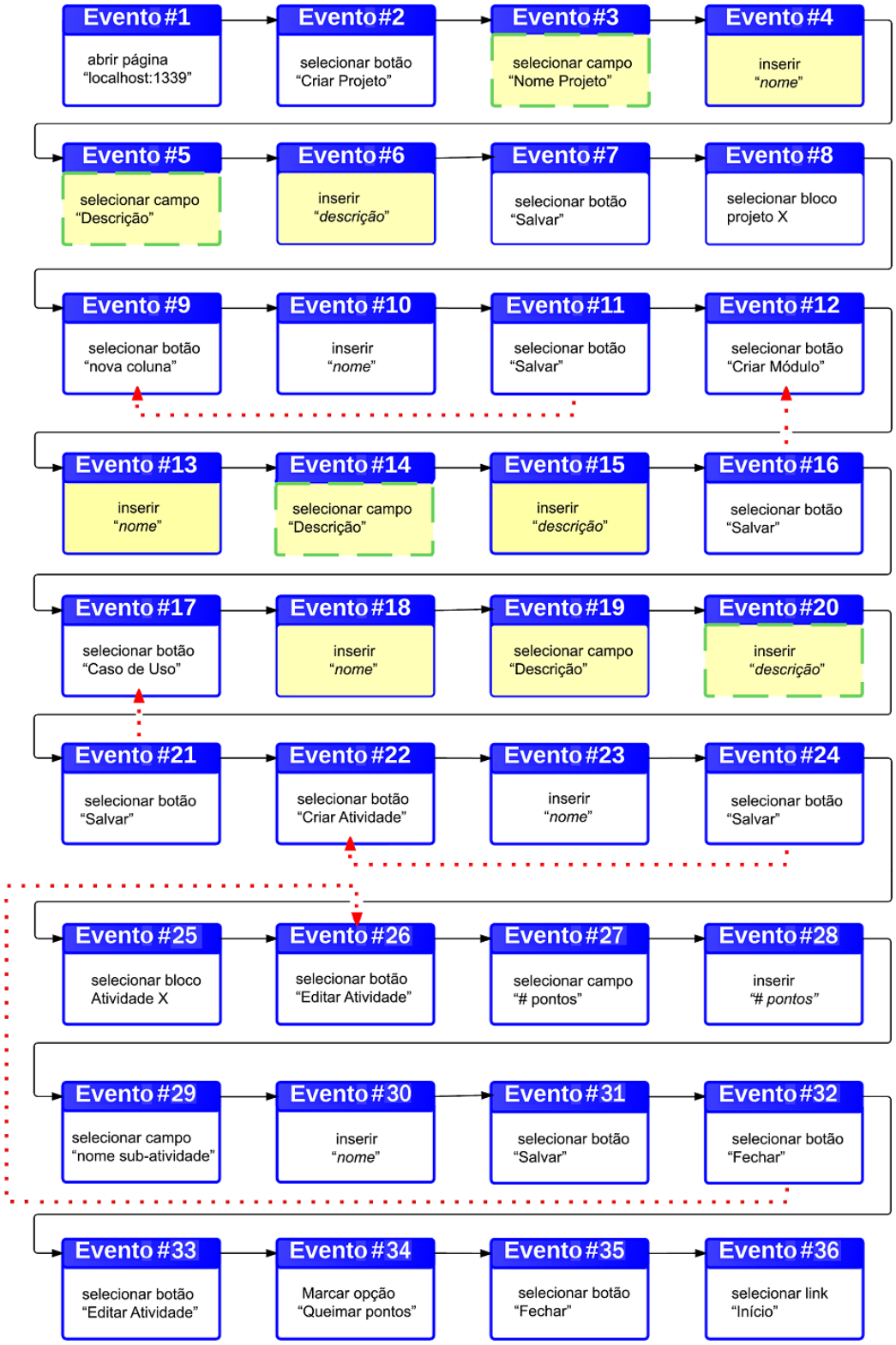


Figura 32: Grafo com eventos relacionados à tarefa de execução de Projetos.

ciclo e 141.056 casos com três iterações de ciclo.

Para cada requisito funcional, há pelo menos um caso de teste gerado pelo UsaTasker++. Com isso, pode-se afirmar que o sistema obteve êxito na validação dos requisitos dessa aplicação. Além disso, o número de sequências únicas e simples geradas corresponde ao total de variações estipuladas pelo Princípio Fundamental da Contagem e as pelas respectivas permutações. Adiciona-se a esse número as permutações com todos os ciclos encontrados. Com isso, pode-se afirmar que o sistema obteve sucesso em gerar todos os caminhos válidos originados da tarefa base.

O sistema executando em uma máquina com processador Intel i5, CPU de 1.80GHz e 4GB de RAM, demorou menos de 20 segundos para listar todos os casos de teste da primeira linha da Tabela 7 (vide Figura 33). Durante a execução, o pico de utilização do processador chegou a 70%. A utilização da memória ficou em torno de 500MB.

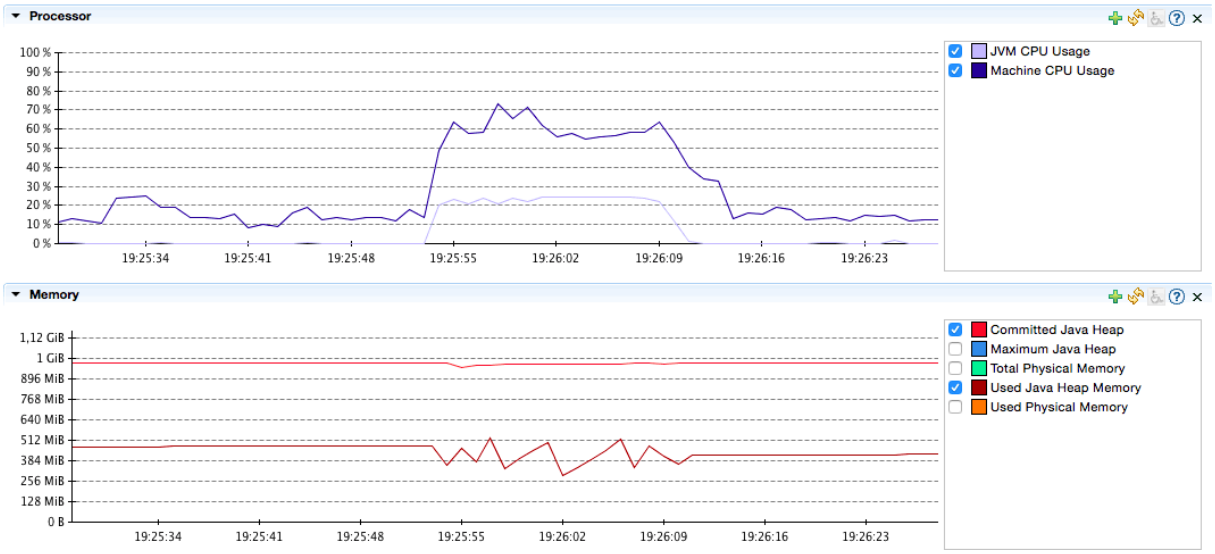


Figura 33: Tempo de execução e memória utilizada para listar 48.640 casos de teste

Já para os 141.056 casos de teste da segunda linha da Tabela 7 foi necessário aumentar a memória dinâmica da Máquina Virtual Java de 512MB para 1.024MB (o pico de utilização da memória foi por volta de 768MB). Além disso, o tempo total gasto para listar todos os casos foi de aproximadamente 400 segundos, com picos de utilização do processador acima de 90% (Figuras 34 e 35).

Os primeiros testes dessa sequência a serem executados foram com apenas uma iteração de ciclo, relativos aos 48.640 casos. Nas primeiras 24 horas de execução apenas 1.545 casos foram finalizados. Desses, 56 testes falharam, todos por motivo de infraestrutura: o processador não suportou a velocidade da automação e executou comandos fora da sequência solicitada. Esse cenário passou a acontecer após o milésimo teste, onde a carga de dados a ser carregada do banco de dados e processada pelo navegador já estava grande

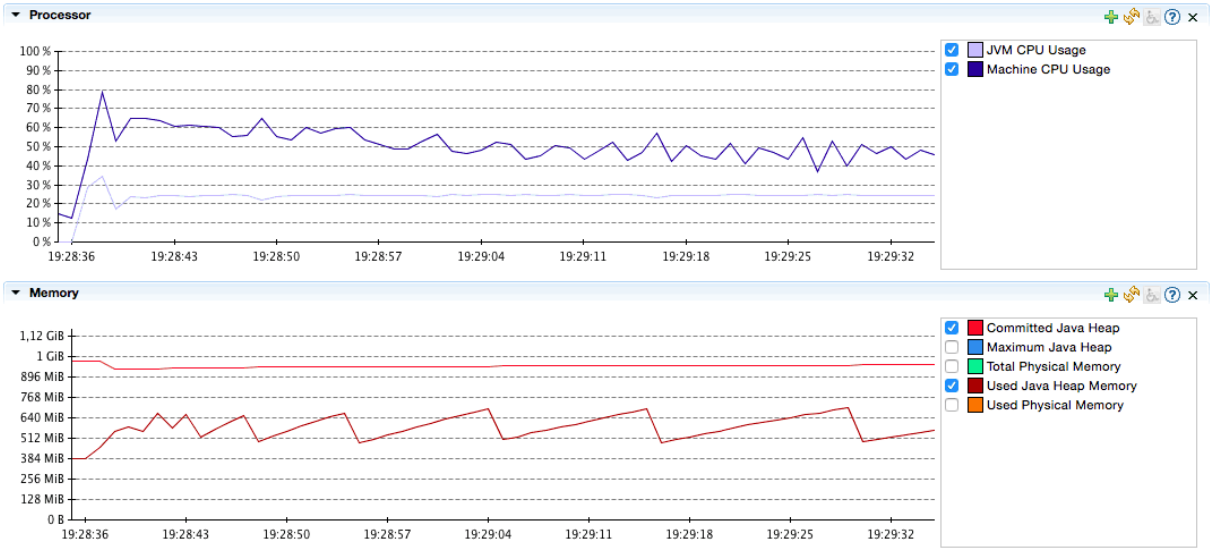


Figura 34: Tempo de execução e memória utilizada no início da geração dos 141.056 casos de teste

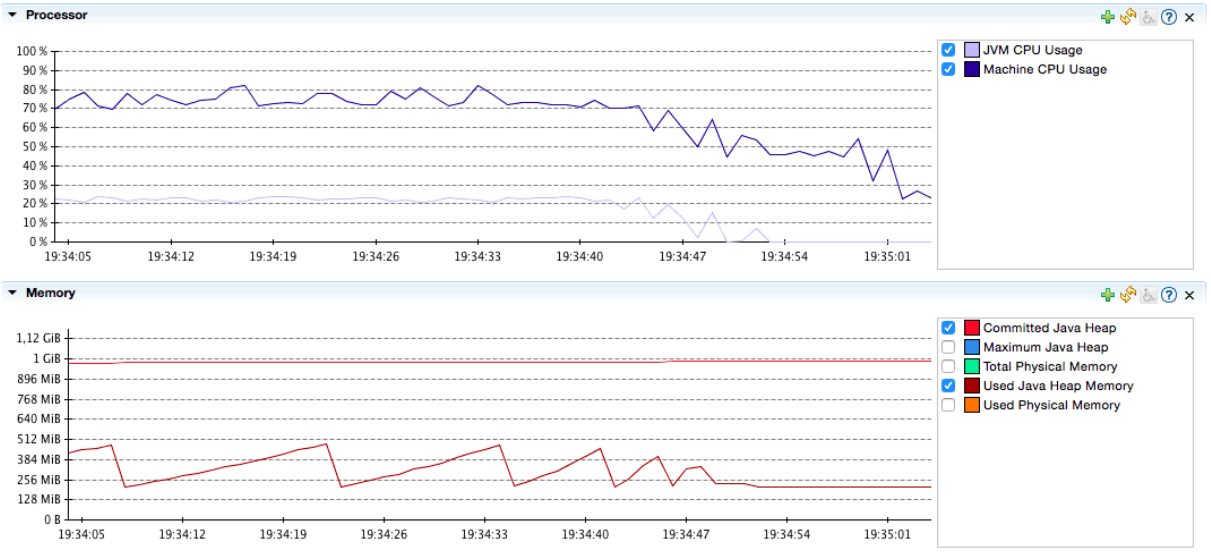


Figura 35: Tempo de execução e memória utilizada no final da geração dos 141.056 casos de teste

(acima de 2GB). A lentidão aumentou até o sistema travar. Os problemas aconteceram em todas as tentativas de execução, mesmo após aumentar o número de *cores* da CPU onde o SUT foi instalado (de 1 para 4).

Apesar dos problemas encontrados, a etapa foi bastante positiva por ser considerada como um teste de carga e estresse. Foi possível identificar problemas na configuração do servidor de aplicação, além de estabelecer os requisitos mínimos do mesmo para que o sistema suporte a capacidade esperada (ao menos 141.000 projetos, considerando que um novo projeto deve ser criado em cada um dos 141.056 cenários). Além disso, os 1.489 casos

de teste que passaram demonstraram que a aplicação efetivamente suporta os requisitos funcionais determinados. Ao menos um teste passou para cada requisito.

Devido às limitações físicas, foi necessário reduzir o número de configurações no cenário, o que, consequentemente, reduziu o número de casos de teste. O desafio era realizar a redução sem diminuir a cobertura dos requisitos funcionais. Ressalta-se que a redução de casos de teste acontece devido à simplificação realizada sobre o cenário. Tal operação não compromete os resultados desse experimento visto que, todos os casos de teste possíveis continuam sendo gerados para os respectivos cenários. Ao simplificar o cenário, o número de testes diminui.

No novo mapeamento, todos os cliques de tela com o objetivo de mudança de foco entre os elementos da interface gráfica foram removidos. Com isso, as sequências de elementos fora de ordem e opcionais foram simplificadas. O número de casos de teste para a sequência definida na Figura 36 foi de 270 para uma iteração de ciclo e 756 para três iterações de ciclo (Tabela 8).

Tabela 8: Testes gerados da terceira sequência

Abordagem	**CI**	**FDO**	**Opc**	**Cc**	**Total**
Completo 3	1	8	8	254	270
Completo 3	3	8	8	740	756

CI: número de iterações dos ciclos; FDO: fora de ordem; Opc: opcional; Cc: caminhos com ciclo.

A primeira bateria de testes levou pouco mais de 2 horas e 5 minutos para ser executada. Já os 756 cenários foram executados em 5 horas e 32 minutos. Todos os casos de teste passaram sem qualquer impacto significativo no servidor de aplicações. Não obstante, o sistema garantiu o bom comportamento dos requisitos funcionais.

5.3 Análise dos estudos de caso

Durante os testes, percebeu-se que a atividade para definir uma sequência básica de eventos, que seja efetiva na validação da aplicação, é extremamente importante. A sequência deve ser completa o suficiente para verificar todos os requisitos funcionais, porém, simples o suficiente a fim de não gerar testes desnecessários que oneram a etapa de verificação.

Além disso, foi possível observar que, conforme discutido na Seção 3.3, executar automaticamente todos os casos de teste possíveis pode ser uma tarefa demorada, custosa

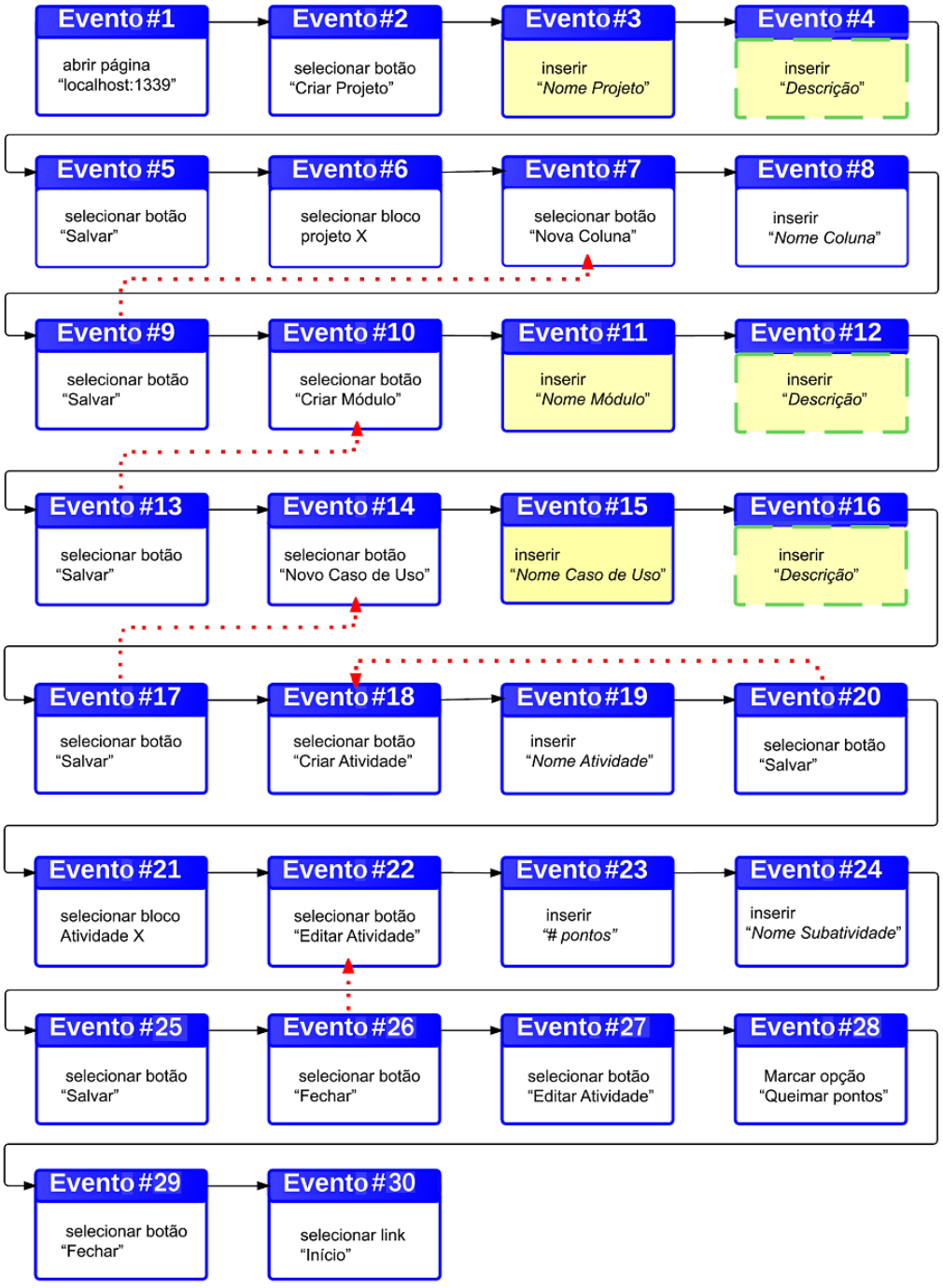


Figura 36: Nova sequência de eventos simplificada para execução de Projetos.

e, até mesmo, inviável. Neste caso, o recomendado é simplificar a sequência básica de eventos para diminuir o número de casos gerados, conforme realizado na última abordagem da Seção 5.2. Outra alternativa é escolher alguns dos cenários criados para realizar a validação manual.

Por fim, evidenciou-se que utilizar o modelo de tarefas para gerar todos os casos de teste pode ser uma alternativa para solucionar os problemas relacionados ao custo de atu-

alização dos casos de teste. Conforme Hauptmann e Junker (HAUPTMANN; JUNKER, 2011), a manutenção dos cenários é custosa em sistemas com mudanças recorrentes nos objetos da interface gráfica, obrigando a atualização de todos os testes que contemplam o objeto modificado. Porém, na abordagem do UsaTasker++, a atualização do objeto acontece somente no modelo de tarefas. Neste caso, a ferramenta processa o modelo e gera os casos de teste já com as atualizações necessárias.

5.4 Considerações finais

Neste capítulo foram realizados experimentos em duas aplicações Web para validar a metodologia de testes implementada no UsaTasker++. O objetivo foi demonstrar que o modelo é capaz de verificar os requisitos funcionais dessas aplicações, além de identificar os possíveis cenários com erros.

Durante os testes da primeira aplicação – um sistema para financiamento de automóveis, o UsaTasker++ obteve sucesso em gerar todos os casos de teste para a sequência de eventos definida. Os 4.160 cenários simples e 12.352 cenários com ciclos foram capazes de validar os requisitos funcionais da aplicação. Neste caso, não foram encontrados erros durante a execução dos testes automatizados.

Na validação da segunda aplicação – um sistema de gerenciamento de projetos ágeis, foi possível gerar 48.640 testes com uma iteração de ciclo e 141.056 testes com três iterações de ciclo. Apesar dos cenários validarem os requisitos funcionais da aplicação, o grande número de testes gerados tornou a execução de todos os testes automatizados inviável. Mesmo assim, a execução serviu como teste de carga e foi capaz de identificar problemas na infraestrutura do servidor de aplicações.

Depois de reestruturar as configurações do cenário básico, o UsaTasker++ atingiu os objetivos esperados, gerando todos os 270 testes automatizados possíveis, os quais validaram os requisitos funcionais da aplicação após pouco mais de duas horas de execução.

Por fim, ficou evidente que, apesar do sistema UsaTasker++ ter validado corretamente as aplicações Web de uma forma simples e ágil, existem algumas limitações quanto a capacidade do desenvolvedor em definir uma sequência básica de eventos e quanto ao número de casos de teste gerados. Uma sequência mal definida pode não abordar todos os requisitos funcionais da aplicação, enquanto pode ser inviável executar automaticamente um número muito grande de cenários de teste. Por outro lado, o sistema é capaz de tratar modificações recorrentes na interface gráfica, atualizando todos os casos de teste

com pouco esforço.

6 Conclusão

O grande número de usuários e o valor estratégico oferecido pelas aplicações Web tornam a verificação dos requisitos funcionais e não funcionais um processo extremamente importante. Apesar de existirem diversas abordagens que auxiliam na etapa de verificacão de *softwares*, nenhuma delas se tornou genérica e eficaz o bastante para tratar sistemas inteiros sem muito esforço. Por isso, no sentido de prover uma ferramenta robusta, capaz de oferecer maior flexibilidade para a avaliação de aplicações Web, foi desenvolvido o UsaTasker++, uma ferramenta para geração de casos de teste baseados em modelo de tarefa.

As diversas ferramentas reportadas na literatura para geração de casos de teste não possuem foco na validação de requisitos funcionais. As poucas alternativas, discutidas e detalhadas na Seção 2.4, apresentam limitações. Essas limitações tornam-se cruciais quando se trata de aplicações Web grandes e dinâmicas.

Este trabalho apresentou os principais conceitos relacionados a testes em aplicações Web. Além disso, a ferramenta USABILICS (VASCONCELOS; BALDOCHI JR., 2011, 2012a, 2012b) foi brevemente descrita para demonstrar as oportunidades originadas a partir das funções oferecidas por tal ferramenta. Apesar das pesquisas anteriores estarem relacionadas com testes de usabilidade, as oportunidades vislumbradas proporcionaram avanços na área de testes funcionais.

Com o intuito de expandir a abordagem de testes oferecida pelo USABILICS, o UsaTasker++ foi desenvolvido. A aplicação é um complemento para o sistema na geração de casos de teste baseados no modelo de tarefas e na execução automática dos mesmos.

A facilidade para criar um cenário a ser verificado, por meio do USABILICS, simplifica o processo global de validação. O sistema permite ao operador gravar as ações diretamente na aplicação Web real. Além disso, é possível indicar as característica de cada evento gravado como *opcional, fora de ordem* e *cíclico*. Após a classificação dos eventos, o UsaTasker++ processa o grafo básico gerado e, utilizando as diversas técni-

cas apresentadas na Seção 3.2, é capaz de gerar todos os casos de teste para o cenário especificado. Por fim, o sistema dispara os testes automatizados referentes a cada caso e apresenta os cenários com erros.

6.1 Resultados obtidos

Analisando os resultados dos experimentos realizados com diferentes aplicações Web, é possível concluir que a abordagem proposta neste trabalho contribuiu de forma significativa para a validação dos requisitos funcionais desses sistemas. Os cenários com erro foram identificados e os cenários válidos foram executados com sucesso.

Além disso, os experimentos mostram que os casos de teste gerados são todos os cenários possíveis para o fluxo definido. As classificações *opcional, fora de ordem* e *cíclico* determinam um grande número de casos diferentes, porém, quanto maior o número de classificações em uma mesma sequência de eventos, maior o número de cenários a serem criados.

Uma das principais conclusões deste trabalho é que o sistema desenvolvido é eficaz na criação de testes que validam os requisitos funcionais de um sistema Web, porém, para que esta etapa seja bem sucedida, o analista de testes responsável pela criação do fluxo base deve ter conhecimento avançado do sistema de forma a evitar que o número de casos de teste gerados seja excessivo e até mesmo inviável. Mesmo assim, por gerar uma grande quantidade de casos de teste, o sistema também demostrou ter aplicabilidade para testes de carga, estresse e regressão.

Por fim, a ferramenta desenvolvida pode ser considerada como uma plataforma de testes mais completa e robusta, visto que ela abrange não só a avaliação da usabilidade como também verifica os requisitos funcionais das aplicações Web.

6.2 Trabalhos futuros

Espera-se que as contribuições produzidas neste trabalho possam reduzir o esforço necessário na etapa de testes, especialmente em grande aplicações comerciais, fomentando as pesquisas e os avanços das técnicas de validação e verificação de aplicações Web. Com este intuito, e também para contribuir com a evolução dos trabalhos existentes, são apresentadas as seguintes sugestões para trabalhos futuros:

1. Utilizar técnicas para definição de um número otimizado de casos de teste, como a apresentada por Hirzel (HIRZEL, 2014), a fim economizar os recursos necessários para execução dos testes automatizados;

2. Implementar uma interface para configuração de elementos HTML com identificadores dinâmicos, habilitando um arquivo de configuração e eliminando a necessidade da compilação do código após o mapeamento;

3. Utilizar o modelo proposto para execução de testes de carga e estresse para facilitar a implementação de recursos característicos desses tipos de testes.

7 Apêndice A – Disponibilidade da ferramenta

A ferramenta detalhada neste trabalho é disponível para pesquisa e pode ser utilizada por meio dos seguintes passos:

1. instalar um servidor de aplicações Java (recomenda-se o *Glassfish* ou *Tomcat*);
2. instalar o servidor de banco de dados *Postgres*;
3. realizar o *download* do código no GitLab[1]
4. armazenar o código do USABILICS no servidor de aplicações;
5. atualizar o código do USABILICS com os arquivos relacionados ao UsaTasker++;
6. compilar e instalar o sistema no servidor de aplicações;
7. configurar o USABILICS conforme instruções do arquivo LEIA-ME;
8. acessar o USABILICS pelo navegador;
9. cadastrar uma tarefa para uma aplicação Web e gravar os passos via USABILICS;
10. acessar os detalhes da tarefa e selecionar a opção "Gerar Testes";

A tela do UsaTasker++, referenciada pela opção "Gerar Testes", contempla a geração dos casos de teste e a execução dos testes automatizados.

[1] *https://gitlab.com/flavio-rezende/USABILICS*

Referências

AL-ZAIN, S.; ELEYAN, D.; GARFIELD, J. Automated user interface testing for web applications and test complete. In: *Proceedings of the CUBE International Information Technology Conference*. New York, NY, USA: ACM, 2012. (CUBE '12), p. 350–354. ISBN 978-1-4503-1185-4. Disponível em: <http://doi.acm.org/10.1145/2381716.2381782>.

ALALFI, M. H.; CORDY, J. R.; DEAN, T. R. Modelling methods for web application verification and testing: State of the art. *Softw. Test. Verif. Reliab.*, John Wiley and Sons Ltd., Chichester, UK, v. 19, n. 4, p. 265–296, dez. 2009. ISSN 0960-0833. Disponível em: <http://dx.doi.org/10.1002/stvr.v19:4>.

ALFARO, L. de; HENZINGER, T. A.; MANG, F. Y. C. Mcweb: A model-checking tool for web site debugging. In: *In World Wide Web Conference – Poster Proceedings*. [S.l.: s.n.], 2001. (WWW '01), p. 86–87.

ALSHAHWAN, N.; HARMAN, M. Automated web application testing using search based software engineering. In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2011. (ASE '11), p. 3–12. ISBN 978-1-4577-1638-6. Disponível em: <http://dx.doi.org/10.1109/ASE.2011.6100082>.

ANDREWS, A. A.; OFFUTT, J.; ALEXANDER, R. T. Testing web applications by modeling with fsms. *Software and Systems Modeling*, v. 4, p. 326–345, 2005.

ARORA, A.; SINHA, M. Web application testing: A review on techniques, tools and state of art. *International Journal of Scientific & Engineering Research*, IJSER, v. 3, n. 2, p. 1–6, feb 2012. ISSN 2229-5518.

BARBOSA, A.; PAIVA, A. C.; CAMPOS, J. C. Test case generation from mutated task models. In: *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York, NY, USA: ACM, 2011. (EICS '11), p. 175–184. ISBN 978-1-4503-0670-6. Disponível em: <http://doi.acm.org/10.1145/1996461.1996516>.

BAU, J.; BURSZTEIN, E.; GUPTA, D.; MITCHELL, J. State of the art: Automated black-box web application vulnerability testing. In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2010. (SP '10), p. 332–345. ISBN 978-0-7695-4035-1. Disponível em: <http://dx.doi.org/10.1109/SP.2010.27>.

BAUERSFELD, S.; WAPPLER, S.; WEGENER, J. An approach to automatic input sequence generation for gui testing using ant colony optimization. In: *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2011. (GECCO '11), p. 251–252. ISBN 978-1-4503-0690-4. Disponível em: <http://doi.acm.org/10.1145/2001858.2001999>.

BENEDIKT, M.; FREIRE, J.; GODEFROID, P. Veriweb: Automatically testing dynamic web sites. In: *Proceedings of the Eleventh International World Wide Web Conference*. [S.l.: s.n.], 2002. (WWW '02).

BERNER, S.; WEBER, R.; KELLER, R. K. Observations and lessons learned from automated testing. In: *Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: ACM, 2005. (ICSE '05), p. 571–579. ISBN 1-58113-963-2. Disponível em: <http://doi.acm.org/10.1145/1062455.1062556>.

BRETON, G. L.; MARONNAUD, F.; HALLÉ, S. Automated exploration and analysis of ajax web applications with webmole. In: *Proceedings of the 22Nd International Conference on World Wide Web Companion*. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2013. (WWW '13 Companion), p. 245–248. ISBN 978-1-4503-2038-2. Disponível em: <http://dl.acm.org/citation.cfm?id=2487788.2487913>.

BRIAND, L. C.; LABICHE, Y.; SHOUSHA, M. Stress testing real-time systems with genetic algorithms. In: *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2005. (GECCO '05), p. 1021–1028. ISBN 1-59593-010-8. Disponível em: <http://doi.acm.org/10.1145/1068009.1068183>.

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. Disponível em: <http://doi.acm.org/10.1145/1321631.1321681>.

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. Disponível em: <http://doi.acm.org/10.1145/2103380.2103421>.

COLLINS, E. F.; LUCENA JR., V. F. de. Software test automation practices in agile development environment: An industry experience report. In: *Proceedings of the 7th International Workshop on Automation of Software Test*. Piscataway, NJ, USA: IEEE Press, 2012. (AST '12), p. 57–63. ISBN 978-1-4673-1822-8. Disponível em: <http://dl.acm.org/citation.cfm?id=2663608.2663620>.

DOGAN, S.; BETIN-CAN, A.; GAROUSI, V. Web application testing: A systematic literature review. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 91, p. 174–201, maio 2014. ISSN 0164-1212. Disponível em: <http://dx.doi.org/10.1016/j.jss.2014.01.010>.

ELBAUM, S.; KARRE, S.; ROTHERMEL, G. Improving web application testing with user session data. In: *Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003. (ICSE '03), p. 49–59. ISBN 0-7695-1877-X. Disponível em: <http://dl.acm.org/citation.cfm?id=776816.776823>.

ELBAUM, S.; ROTHERMEL, G.; KARRE, S.; II, M. F. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, IEEE Press,

Piscataway, NJ, USA, v. 31, n. 3, p. 187–202, mar 2005. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2005.36>.

FORBES, I. *The Reputational Impact of IT Risk*. 2014. <http://www-935.ibm.com/services/multimedia/RLL12363USEN_2014_Forbes_Insights.pdf>. [Online; accessed 04-June-2015].

FRANTZEN, L.; HUERTA, M. L. N.; KISS, Z. G.; WALLET, T. Web services and formal methods. In: BRUNI, R.; WOLF, K. (Ed.). Berlin, Heidelberg: Springer-Verlag, 2009. cap. On-The-Fly Model-Based Testing of Web Services with Jambition, p. 143–157. ISBN 978-3-642-01363-8.

GIAS, A. U.; SAKIB, K. An adaptive bayesian approach for url selection to test performance of large scale web-based systems. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014. (ICSE Companion 2014), p. 608–609. ISBN 978-1-4503-2768-8. Disponível em: <http://doi.acm.org/10.1145/2591062.2591139>.

GODEFROID, P.; KLARLUND, N.; SEN, K. Dart: Directed automated random testing. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 40, n. 6, p. 213–223, jun 2005. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1064978.1065036>.

HARMAN, M.; JONES, B. F. The seminal workshop: Reformulating software engineering as a metaheuristic search problem. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 26, n. 6, p. 62–66, nov 2001. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/505532.505548>.

HAUPTMANN, B.; JUNKER, M. Utilizing user interface models for automated instantiation and execution of system tests. In: *Proceedings of the First International Workshop on End-to-End Test Script Engineering*. New York, NY, USA: ACM, 2011. (ETSE '11), p. 8–15. ISBN 978-1-4503-0808-3. Disponível em: <http://doi.acm.org/10.1145/2002931.2002933>.

HIRZEL, M. Selective regression testing for web applications created with google web toolkit. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. New York, NY, USA: ACM, 2014. (PPPJ '14), p. 110–121. ISBN 978-1-4503-2926-2. Disponível em: <http://doi.acm.org/10.1145/2647508.2647527>.

IVORY, M. Y.; HEARST, M. A. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 33, n. 4, p. 470–516, dez. 2001. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/503112.503114>.

JESUS, F. R. de; VASCONCELOS, L. G. de; BALDOCHI, L. A. Leveraging task-based data to support functional testing of web applications. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2015. (SAC '15), p. 783–790. ISBN 978-1-4503-3196-8. Disponível em: <http://doi.acm.org/10.1145/2695664.2695917>.

KAM, B.; DEAN, T. R. Lessons learned from a survey of web applications testing. In: *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*. Washington, DC, USA: IEEE Computer Society, 2009. (ITNG '09), p. 125–130. ISBN 978-0-7695-3596-8. Disponível em: <http://dx.doi.org/10.1109/ITNG.2009.306>.

LEITNER, A.; CIUPA, I.; ORIOL, M.; MEYER, B.; FIVA, A. Contract driven development = test driven development - writing test cases. In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. New York, NY, USA: ACM, 2007. (ESEC-FSE '07), p. 425–434. ISBN 978-1-59593-811-4. Disponível em: <http://doi.acm.org/10.1145/1287624.1287685>.

LEOTTA, M.; CLERISSI, D.; RICCA, F.; SPADARO, C. Comparing the maintainability of selenium webdriver test suites employing different locators: A case study. In: *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation*. New York, NY, USA: ACM, 2013. (JAMAICA 2013), p. 53–58. ISBN 978-1-4503-2161-7. Disponível em: <http://doi.acm.org/10.1145/2489280.2489284>.

LI, Y.-F.; DAS, P. K.; DOWE, D. L. Two decades of web application testing: A survey of recent advances. *Information Systems*, v. 43, n. 0, p. 20 – 54, 2014. ISSN 0306-4379. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0306437914000271>.

LUCCA, G. A. D.; FASOLINO, A. R. Testing web-based applications: The state of the art and future trends. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 48, n. 12, p. 1172–1186, dez. 2006. ISSN 0950-5849. Disponível em: <http://dx.doi.org/10.1016/j.infsof.2006.06.006>.

MARCHETTO, A.; TONELLA, P.; RICCA, F. State-based testing of ajax web applications. In: *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2008. (ICST '08), p. 121–130. ISBN 978-0-7695-3127-4. Disponível em: <http://dx.doi.org/10.1109/ICST.2008.22>.

MCMASTER, S.; YUAN, X. Developing a feedback-driven automated testing tool for web applications. In: *Proceedings of the 2012 12th International Conference on Quality Software*. Washington, DC, USA: IEEE Computer Society, 2012. (QSIC '12), p. 210–213. ISBN 978-0-7695-4833-3. Disponível em: <http://dx.doi.org/10.1109/QSIC.2012.25>.

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. Disponível em: <http://doi.acm.org/10.1145/2109205.2109208>.

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

NEDYALKOVA, S.; BERNARDINO, J. Open source capture and replay tools comparison. In: *Proceedings of the International C* Conference on Computer Science and*

Software Engineering. New York, NY, USA: ACM, 2013. (C3S2E '13), p. 117–119. ISBN 978-1-4503-1976-8. Disponível em: <http://doi.acm.org/10.1145/2494444.2494464>.

PAPADAKIS, M.; MALEVRIS, N.; KALLIA, M. Towards automating the generation of mutation tests. In: *Proceedings of the 5th Workshop on Automation of Software Test*. New York, NY, USA: ACM, 2010. (AST '10), p. 111–118. ISBN 978-1-60558-970-1. Disponível em: <http://doi.acm.org/10.1145/1808266.1808283>.

PRAPHAMONTRIPONG, U.; OFFUTT, J. Applying mutation testing to web applications. 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. 132–141. ISBN 978-0-7695-4050-4. Disponível em: <http://dx.doi.org/10.1109/ICSTW.2010.38>.

RICCA, F.; TONELLA, P. Analysis and testing of web applications. In: *Proceedings of the 23rd International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2001. (ICSE '01), p. 25–34. ISBN 0-7695-1050-7. Disponível em: <http://dl.acm.org/citation.cfm?id=381473.381476>.

ROSSI, G. Web modeling languages strike back. *IEEE Internet Computing*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 17, n. 4, p. 4–6, jul. 2013. ISSN 1089-7801. Disponível em: <http://dx.doi.org/10.1109/MIC.2013.78>.

SCIASCIO, E. D.; DONINI, F. M.; MONGIELLO, M.; PISCITELLI, G. Anweb: A system for automatic support to web application verification. In: *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*. New York, NY, USA: ACM, 2002. (SEKE '02), p. 609–616. ISBN 1-58113-556-4. Disponível em: <http://doi.acm.org/10.1145/568760.568866>.

SEDGEWICK, R. Permutation generation methods. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 9, n. 2, p. 137–164, jun. 1977. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/356689.356692>.

SELENIUMHQ. *Selenium Web Driver*. 2015. <http://seleniumhq.org/>. "[Online; accessed 12-March-2015]".

SHIROLE, M.; KUMAR, R. Uml behavioral model based test case generation: A survey. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 38, n. 4, p. 1–13, jul. 2013. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/2492248.2492274>.

SPRENKLE, S.; GIBSON, E.; SAMPATH, S.; POLLOCK, L. Automated replay and failure detection for web applications. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2005. (ASE '05), p. 253–262. ISBN 1-58113-993-4. Disponível em: <http://doi.acm.org/10.1145/1101908.1101947>.

THUMMALAPENTA, S.; LAKSHMI, K. V.; SINHA, S.; SINHA, N.; CHANDRA, S. Guided test generation for web applications. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 162–171. ISBN 978-1-4673-3076-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2486788.2486810>.

THUMMALAPENTA, S.; SINHA, S.; SINGHANIA, N.; CHANDRA, S. Automating test automation. In: *Proceedings of the 34th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012. (ICSE '12), p. 881–891. ISBN 978-1-4673-1067-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2337223.2337327>.

VASCONCELOS, L. G. de; BALDOCHI JR., L. A. Usabilics: Avaliação remota de usabilidade e métricas baseadas na análise de tarefas. In: *Proceedings of the 10th Brazilian Symposium on on Human Factors in Computing Systems and the 5th Latin American Conference on Human-Computer Interaction*. Porto Alegre, Brazil, Brazil: Brazilian Computer Society, 2011. (IHC+CLIHC '11), p. 303–312. ISBN 978-85-7669-257-7. Disponível em: <http://dl.acm.org/citation.cfm?id=2254436.2254488>.

VASCONCELOS, L. G. de; BALDOCHI JR., L. A. Towards an automatic evaluation of web applications. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2012a. (SAC '12), p. 709–716. ISBN 978-1-4503-0857-1. Disponível em: <http://doi.acm.org/10.1145/2245276.2245410>.

VASCONCELOS, L. G. de; BALDOCHI JR., L. A. Usatasker: a task definition tool for supporting the usability evaluation of web applications. In: . Madrid, Spain: IADIS, 2012b. p. 307–314.

WANG, F.; DU, W. A test automation framework based on web. In: *Proceedings of the 2012 IEEE/ACIS 11th International Conference on Computer and Information Science*. Washington, DC, USA: IEEE Computer Society, 2012. (ICIS '12), p. 683–687. ISBN 978-0-7695-4694-0. Disponível em: <http://dx.doi.org/10.1109/ICIS.2012.21>.

WANG, W.; SAMPATH, S.; LEI, Y.; KACKER, R.; LAWRENCE, J. A combinatorial approach to building navigation graphs for dynamic web applications. In: *Proceedings of the 2009 International Conference on Software Maintenance*. [S.l.]: IEEE Computer Society, 2009. (ICSM '09), p. 211–220.

YANG, Y.; ZHANG, H.; PAN, M.; YANG, J.; HE, F.; LI, Z. A model-based fuzz framework to the security testing of tcg software stack implementations. In: *Proceedings of the 2009 International Conference on Multimedia Information Networking and Security - Volume 01*. Washington, DC, USA: IEEE Computer Society, 2009. (MINES '09), p. 149–152. ISBN 978-0-7695-3843-3. Disponível em: <http://dx.doi.org/10.1109/MINES.2009.111>.

ZHONG, H.; ZHANG, L.; MEI, H. An experimental study of four typical test suite reduction techniques. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 50, n. 6, p. 534–546, maio 2008. ISSN 0950-5849. Disponível em: <http://dx.doi.org/10.1016/j.infsof.2007.06.003>.