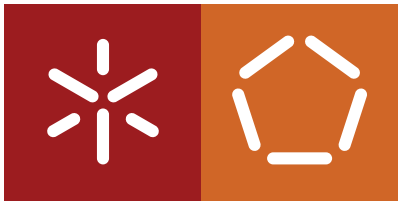

Universidade do Minho
Departamento de Informática

Mestrado em Engenharia Informática

Testes Baseados em Modelos

Autor
Raphael Julien Rodrigues

Orientado por:

José Creissac Campos

Braga, 30 de Abril 2015

Agradecimentos

Gostaria de agradecer e deixar a minha gratidão a todos que estiverem ao meu lado durante a realização deste projecto. Agradecer principalmente ao meu supervisor pela oportunidade que me ofereceu, pela disponibilidade para esclarecer dúvidas, para resolver problemas e por toda a ajuda no desenvolvimento do projecto. Um especial obrigado à minha família e amigos por todo o apoio que me derem ao longo do projecto.

Este trabalho é financiado por Fundos FEDER através do Programa Operacional Factores de Competitividade – COMPETE e por Fundos Nacionais através da FCT – Fundação para a Ciência e a Tecnologia no âmbito do projecto FCOMP-01-0124-FEDER-020554. O autor é ainda suportado por uma bolsa com referência PTDC/EIA-EIA/119479/2010_UMINHO.

Abstract

The graphical user interface (GUI, the English Graphical User Interface) is very important to the success of an interactive system. The malfunction of the GUI can compromise the proper functioning of the application and consequently of the entire software system, which may cause economic losses. It is therefore necessary to ensure system quality. One way to achieve this objective is through software testing. The task of testing a software is a process of executing a set of steps in order to evaluate if the behavior of the system is the expected, in the environment for which it was designed. Currently, applications tend to become increasingly large, complex, sophisticated interactive and other integrated systems, which makes the process of testing an expensive and heavy phase in the development cycle. It is therefore essential to find automated solutions.

Model-based testing is a black-box technique with the objective of verifying if the implementation of a software meets in accordance with its specification. Model-based testing is focused on automatic generation of tests, where these will serve to verify if the implementation meets in accordance with the model.

This thesis will be focused on model-based testing applied graphical user interfaces. The main objective is to automate as much as possible, the generation and execution of tests focused on the graphical user interface.

Resumo

A interface gráfica com o utilizador (GUI, do Inglês Graphical User Interface) é bastante importante para o sucesso de um sistema interactivo. Um incorreto funcionamento da GUI pode inviabilizar o bom funcionamento da aplicação e consequentemente do sistema de software, podendo este mau funcionamento significar perdas económicas. É assim necessário garantir a qualidade do sistema. Uma forma de conseguir isso é através dos testes de software. A tarefa de testar um software é um processo de execução do mesmo seguindo um conjunto de passos para avaliar se o seu comportamento actual é o mais esperado no ambiente para o qual foi projectado. Actualmente, as aplicações tendem a tornar-se cada vez maiores, complexas, sofisticadas, interativas e integradas noutros sistemas, o que torna o processo de testes uma fase bastante cara e pesada no ciclo de desenvolvimento. É por isso, fundamental encontrar soluções automáticas.

O teste baseado em modelos é uma técnica do tipo "caixa preta" (*black box*) que tem como objectivo verificar se a implementação de um software se encontra de acordo com a sua especificação. O teste baseado em modelos foca-se na geração automática de testes, em que estes servirão para verificar se a implementação se encontra de acordo com o modelo.

Esta dissertação irá ser focada nos testes baseados em modelos aplicados a interfaces gráficas com o utilizador, sendo o principal objectivo automatizar, tanto quanto possível, a geração e execução de testes focados na interface.

Conteúdo

Conteúdos iv

Lista de Figuras vii

1 Introdução 4
1.1 Motivação 5
1.2 Estrutura do documento 5

2 Testes de software 7
2.1 Ciclo de vida do teste e diferentes tipos de teste 7
2.1.1 Tipos de testes 8
2.2 Métodos de Testes 11
2.2.1 *White Box Testing* 11
2.2.2 *Black Box Testing* 11
2.3 Testes manuais e automáticos 12
2.4 Avaliação de interfaces gráficas 12
2.4.1 Testes Funcionais 13
2.4.2 Testes de Usabilidade 13
2.5 Construção e execução de testes automáticos para Web 14
2.6 Conclusão 16

3 Testes baseados em modelos 17
3.1 Processo do model-based testing 17
3.1.1 Vantagens 18
3.1.2 Limitações 19
3.2 Model-based testing em interfaces gráficas 20
3.2.1 Trabalhos relacionados 20

3.3 Mutações . . . 21
3.3.1 Testes positivos e negativos . . . 21
3.3.2 Mutações . . . 22
3.3.3 Tipos de Mutações implementadas . . . 22
3.4 Conclusão . . . 23

4 Modelação de comportamento 24
4.1 Máquinas de estado . . . 24
4.2 Utilização no âmbito dos testes . . . 25
4.3 SCXML . . . 25
4.3.1 SCXML como modelo de interfaces gráficas . . . 26
4.4 Exemplo de modelação de uma interface gráfica em SCXML . . . 28
4.4.1 Representação em SCXML . . . 28
4.4.2 Exemplos . . . 29
4.5 Conclusão . . . 33

5 Uma abordagem para testes baseados em modelos 34
5.1 Abordagem proposta . . . 34
5.2 Modelo . . . 36
5.3 Ficheiros de Configuração . . . 36
5.3.1 Demonstração . . . 37
5.3.2 Catálogo de validações possíveis . . . 39
5.4 Transformação do modelo para grafo . . . 40
5.5 Geração de testes abstractos . . . 42
5.5.1 Critérios de cobertura . . . 42
5.6 Geração de casos de teste executáveis . . . 43
5.6.1 Estrutura dos casos de teste . . . 45
5.6.2 Configuração da geração de casos de teste . . . 45
5.7 Execução dos Testes e Avaliação . . . 46
5.8 Mutações . . . 47
5.9 Conclusão . . . 50

6 Implementação da abordagem para testes baseados em modelos 52
6.1 Arquitectura da solução . . . 52

6.2 Packages . . . 53
6.2.1 *Package* graph_elements . . . 53
6.2.2 *Package parser_model* . . . 54
6.2.3 *Package parser_configurations* . . . 54
6.2.4 *Package graph_traverssal* . . . 54
6.2.5 *Package code_generator* . . . 55
6.2.6 *Package mutations* . . . 56
6.2.7 *Package util* . . . 56

7 Exemplos de aplicação 57
7.1 AgroSocial . . . 57
7.1.1 Modelo . . . 57
7.1.2 Ficheiros de Configuração . . . 60
7.1.3 Configuração . . . 61
7.1.4 Geração de casos de teste abstractos . . . 61
7.1.5 Geração de casos de teste executáveis . . . 62
7.1.6 Execução e Avaliação . . . 63
7.1.7 Mutações . . . 65
7.1.8 Erro detectado . . . 66
7.2 Twitter . . . 67
7.2.1 Modelação de pedidos assíncronos . . . 69
7.2.2 Estatísticas . . . 70
7.2.3 Análise . . . 71
7.3 Conclusão . . . 71

8 Conclusões 72
8.1 Objectivos Alcançados e Limitações . . . 72
8.2 Trabalho Futuro . . . 73

Bibliografia 75

A Casos de Estudo 79
A.1 Caso de estudo do Agrosocial . . . 79
A.2 Caso de estudo do Twitter . . . 92

Lista de Figuras

2.1 Modelo em V (adaptado de [1]) . . . 9

3.1 Processo de Model based Testing (adaptado de [2]) . . . 18

4.1 Representação gráfica de uma máquina de estados . . . 26
4.2 Máquina de estados . . . 30
4.3 Mapeamento entre implementação e modelo . . . 32

5.1 Arquitectura do processo de testes . . . 35
5.2 Código gerado a partir dos ficheiros de configuração . . . 39
5.3 Exemplo de uma validação . . . 40
5.4 Grafo parcial gerado a partir do modelo . . . 41
5.5 Caminho de um caso de teste . . . 42
5.6 Execução do teste para o formulário de voos . . . 44
5.7 Log do cleartrip . . . 47
5.8 Representação do formulário de pesquisa de voos e respectivo estado de erro . . . 50
5.9 Estado do formulário depois de ter sido feita a mutação . . . 51

6.1 Organização de classes e packages . . . 53

7.1 Máquina de estados AgroSocial . . . 58
7.2 Interface gráfica para criação de novo anúncio e modelo em SCXML para a mesma . . . 59
7.3 Parte dos ficheiros de configuração (mapeamento e dados) para o AgroSocial . . . 60
7.4 Caminhos encontrados . . . 62
7.5 Interface gráfica para criar um novo anúncio e respectivo código gerado . . . 63
7.6 Resultado da criação de um novo anúncio e validações . . . 64
7.7 Log do teste . . . 65
7.8 Exemplo de código gerado com uma mutação . . . 67

7.9 Resultado da submissão do formulário com mutações 68
7.10 Relatório de teste 68
7.11 Representação da maquina de estados para o Twitter 69
7.12 Casos de teste gerados para o Twitter com variações no algoritmo 70

A.1 Caminho abstracto encontrado 79

1 | Introdução

Actualmente grande parte das empresas e instituições precisam de ter os seus produtos e serviços disponíveis em qualquer lugar do mundo e a qualquer hora. As aplicações Web desempenham um papel crucial para atingir esse objectivo. Sendo estas instituições cada vez mais dependentes de sistemas de software para o seu funcionamento, torna-se indispensável criar aplicações Web de elevada qualidade. E com a crescente complexidade das mesmas, existe uma necessidade crescente de automatização do processo de testes.

Uma aplicação Web é um conjunto de páginas, podendo estas ser dinamicamente geradas, que fornecem um serviço para os seus utilizadores. Tipicamente são construídas com arquitetura com três camadas: camada de interface com utilizador, camada de lógica de negócio e uma camada de dados. A camada de interface interage diretamente com o utilizador realizando pedidos sobre a camada lógica de forma a executar as funcionalidades do sistema. Por isso acaba por ser fundamental para a comunicação entre o utilizador e o sistema.

Um incorreto funcionamento da GUI (*Graphical User Interface*) pode inviabilizar o bom funcionamento da aplicação e consequentemente do sistema de software, podendo este mau funcionamento significar perdas económicas. É assim necessário garantir a qualidade do sistema. Uma das formas para o fazer é através dos testes de software. Os testes de software são um processo para revelar erros, sendo úteis para ganhar confiança na implementação e em que esta cumpre os requisitos e especificação [3].

Atualmente, as aplicações tendem a tornar-se cada vez maiores, complexas, sofisticadas, interativas e integradas noutros sistemas, o que torna o processo de testes uma fase bastante cara e pesada no ciclo de desenvolvimento, sendo que os testes manuais acabam por se mostrar demorados, executados poucas vezes e acabando por cobrir poucos casos. Por sua vez, os testes automáticos, são mais rápidos e cobrem mais casos, apesar de possuírem um âmbito de aplicação mais restrito (ou seja, cada técnica tende a testar especificamente um tipo de erro).

1.1 Motivação

O presente trabalho foi desenvolvido no âmbito do projeto "PBGT - Pattern Based GUI Testing". Este projecto tem como objectivo a realização de pesquisas sobre o desenvolvimento e validação de estratégias de teste baseado em modelos sobre interfaces gráficas. Os testes baseados em modelos (MBT – do inglês *Model Based Testing*) são umas das técnicas de teste que permite sistematizar e automatizar o processo de teste, comparando o sistema implementado com um modelo do sistema (ver, por exemplo, [4]). Um dos problemas que surge quando se aplica o MBT a interfaces gráficas é ser necessário fazer a tradução entre eventos no modelo e acções concretas na interface gráfica e entre o que acontece na interface e os estados do modelo. É importante desenvolver um ambiente para a geração e execução de casos de teste de modo a criar GUI's com um maior grau de confiabilidade e com mais qualidade para o utilizador final.

O ambiente desenvolvido deve automatizar o processo de testes aplicados a interfaces gráficas abrangendo todo o processo de teste baseado em modelos, desde a criação do modelo até à produção de relatórios do resultado dos casos de teste gerados, depois de executados. Para a execução dos testes, o trabalho considera o caso particular das aplicações Web.

Pretende-se com esta dissertação desenvolver uma ferramenta capaz de, a partir de um modelo da GUI, gerar automaticamente casos de teste para o sistema que vai ser testado. Estes testes, gerados de forma automática, descrevem a normal interação entre utilizadores e o sistema. No entanto, por vezes os utilizadores podem cometer erros e ter comportamentos inesperados. Por isso, é necessário criar casos de teste que simulem esses comportamentos de forma a saber qual a reacção do sistema aos mesmos. No final os testes deverão ser executados na GUI e os resultados apresentados.

Os principais objectivos desta dissertação são:

- desenvolver uma ferramenta que a partir de uma modelo seja capaz de gerar casos de teste.
- estudar de como incluir erros do utilizador no processo de teste

1.2 Estrutura do documento

- Capítulo 1 - **Introdução** - Contêm uma descrição do projecto e qual o problema que vai tentar resolver.
- Capítulo 2 - **Testes de software** - Apresenta uma introdução ao tema de testes de software para contextualizar o projecto.
- Capítulo 3 - **Testes baseados em modelos** - Explica o que são testes baseados em modelos e qual o seu processo.

- Capítulo 4 - **Modelação de comportamento** - Apresenta as várias formas de modelar o comportamento de uma interface gráfica.
- Capítulo 5 - **Uma abordagem para testes baseados em modelos** - Apresenta o trabalho realizado ao longo do projecto.
- Capítulo 6 - **Implementação da abordagem para testes baseados em modelos** - Apresenta a arquitectura e alguns detalhes técnicos da ferramenta.
- Capítulo 7 - **Demonstração de caso de estudo** - Apresenta casos de estudo usando a ferramenta desenvolvida.
- Capítulo 8 - **Conclusões** - Descreve todo o trabalho realizado, objectivos alcançados e trabalho futuro.

2 | Testes de software

Na engenharia de software actual, a qualidade é um factor que possui cada vez mais relevância, tornando os testes de software uma fase importante no ciclo de desenvolvimento. "O teste é uma atividade realizada para avaliar a qualidade do produto, e para melhorá-lo, através da identificação de defeitos e problemas" [5].

A tarefa de testar um software é um processo de execução do mesmo, seguindo um conjunto de passos para avaliar se o seu comportamento actual é o esperado no ambiente para o qual foi projectado. Os testes de software são importantes para detectar falhas e para a validação do sistema. "Um bom caso de teste é aquele que tem uma alta probabilidade de detectar um erro desconhecido"[6], no entanto um software testado não é um software livre de erros. Devido à complexidade de alguns sistemas torna-se difícil testar todos os caminhos possíveis que o software possui.

Os conceitos de validação e verificação são importantes no que diz respeito à qualidade do software desenvolvido. Os testes de software tanto podem ser de validação (processo de avaliação de um sistema para determinar se satisfaz os requisitos especificados, i.e, assegura que produto final está de acordo com as expectativas do cliente) ou de verificação (assegura que a implementação do produto satisfaz as suas especificações) [7]. No caso de um engenheiro de testes que, a partir dos requisitos do sistema gera um conjunto de testes, estes serão de testes de validação. Se os testes são gerados automaticamente a partir de uma representação abstracta, neste caso então os testes são para verificação [8].

2.1 Ciclo de vida do teste e diferentes tipos de teste

O ciclo de vida do teste (STLC - *Software Testing Life Cycle*) é um conceito fundamental para perceber o processo de testes de um sistema de software em detalhe. O STLC caracteriza cada uma das fases do processo de testes, essenciais para o controlo da qualidade de um software [9]:

1. **Análise/Revisão de requisitos**

   - Identificar os tipos de testes que serão efectuados.

2. **Planeamento dos testes**

   - Determinar custos de testes.
   - Definir ferramentas de testes a utilizar.
   - Preparação do documento do plano de teste / estratégia para vários tipos de testes.

3. **Design dos casos de teste**

   - Criar e validar os casos de teste.
   - Criar os dados para os testes.

4. **Configuração do ambiente de testes**

   - Compreender a arquitectura do ambiente de teste.
   - Configurar o ambiente de teste e dados para teste.

5. **Execução dos testes**

   - Executar os testes com base nos planos de teste e casos de teste preparados.
   - Analisar resultados dos testes.
   - Analise do *log* para casos de teste que falharam.

6. **Relatório dos testes**

   - Documentar resultados dos testes.
   - Criar relatórios para apresentar aos *stakeholders*.

2.1.1 Tipos de testes

Os testes de software são geralmente realizados em diferentes níveis ao longo do processo de desenvolvimento e manutenção [5]. Existem quatro grandes níveis de testes: unitários, de integração, de sistema e de aceitação [5].

- **Testes unitários** testam um pedaço específico do software, tipicamente feitos com acesso ao código que vai ser testado.

- **Testes de integração** verificam se os componentes da aplicação trabalham em conjunto, ou seja, verificam se diferentes componentes funcionam corretamente integrados entre si. Assim, caso algum erro seja encontrado, ele pode ser isolado e corrigido facilmente. Por exemplo, um teste de integração poderá ser verificar se um email é enviado ou recebido.

Existem duas formas de realizar testes de integração, uma chamada de *bottom-up* e outra *top-down*. Os testes de integração *bottom-up* começa com os módulos do nível hierárquico mais baixo, sendo estes posteriormente integrados para serem testados os módulos que se encontram diretamente acima na hierarquia. Na abordagem *top-down* os módulos de mais alto nível são testados primeiro, e depois sucessivamente decompostos de modo a testar os módulos de nível inferior.

- **Testes de sistema** preocupam-se com o comportamento de todo o sistema. Nesta fase grande parte dos erros funcionais deveriam ter sido identificadas durante a fase de testes unitários e de integração. Os teste de sistema são considerados apropriados para testar requisitos não funcionais do sistema, tais como, segurança, desempenho, confiabilidade e usabilidade.

- **Testes de aceitação** são testes conduzidos para determinar se a implementação vai de encontro aos requisitos.

O modelo em V (Figura 2.1) é uma versão modificada do modelo em cascata, e relaciona cada umas das fases do ciclo de desenvolvimento com um dos níveis de testes.

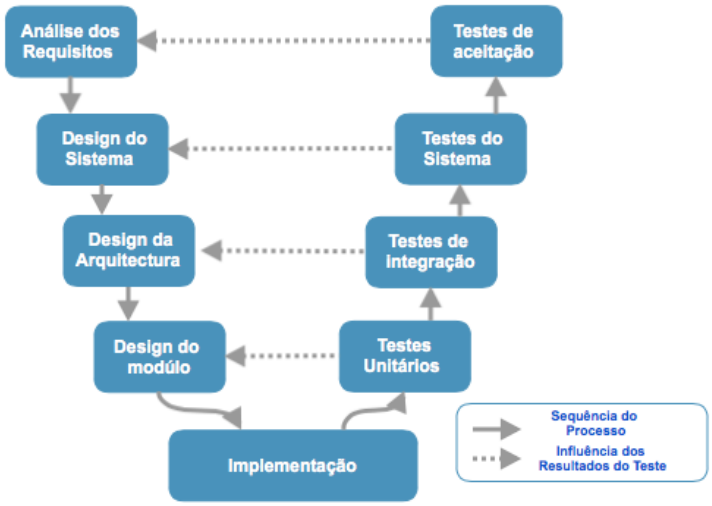

Figura 2.1: Modelo em V (adaptado de [1])

Os testes podem ainda ser classificados relativamente aos diferentes tipos de objectivos que presidem à sua realização:

- **Testes de instalação:** testes realizados no ambiente em que será instalado o sistema.

- **Alpha e beta testing:** Antes do lançamento do software, este é entregue a um pequeno grupo de utilizadores para uso experimental para que estes reportem problemas com o produto. *Alpha testing* é realizado num ambiente controlado. *Beta-Testing* é realizado por utilizadores num ambiente real.

- **Testes funcionais:** visam validar se o comportamento do software testado está em conformidade com as suas especificações.

- **Testes de Regressão:** De acordo com (IEEE610.12-90 [10]), os testes de regressão servem para "re-testar um sistema ou um componente para verificar se as modificações [efectuadas] não causaram efeitos indesejados". Consiste em aplicar, a cada nova versão do software ou a cada ciclo, todos os testes que já foram aplicados nas versões ou ciclos de teste anteriores do sistema.

- **Testes de desempenho:** visam verificar se o software cumpre os requisitos de desempenho, por exemplo, verificar tempos de resposta e disponibilidade do serviço. Verificar o desempenho de um sistema é necessário, sobretudo, no caso dos sistemas de tempo real, uma vez que eles exigem rapidez na resposta do sistema.

- **Testes de Stress:** visam a avaliar o comportamento do sistema quando existem picos de actividades que podem exceder as capacidades do sistema, com intuito de verificar se este falha ou se é capaz de funcionar ou recuperar em tais condições.

- **Back-to-back testing:** consiste em realizar um conjunto de testes em duas versões do software comparando os resultados.

- **Testes de recuperação:** consistem em fazer com que o sistema falhe de diversas maneiras e têm como objectivo verificar as capacidades de recuperação após um "desastre". Esta estratégia deve ser aplicada principalmente em sistemas que devem ser tolerantes a falhas.

- **Testes de configuração:** nos casos em que o software é construído para atender diferentes utilizadores, o teste de configuração analisa o software de acordo com as várias configurações especificadas.

- **Testes de Usabilidade:** avalia o grau com o que utilizador consegue interagir de forma eficaz, eficiente e satisfatória com a aplicação.

- **Test-driven development**: apesar de não ser uma técnica de testes por si só, promove o uso de testes como substituto para um documento de requisitos e não apenas como uma verificação de que o software implementa corretamente os requisitos.

2.2 Métodos de Testes

Para fazer o *design* e realização de testes existem dois tipos de abordagens, *white box* e *black box*. Qualquer uma delas pode ser aplicada durante o processo de testes. No entanto cada uma delas é preferencialmente aplicável a determinados tipos de componentes e com diferentes objectivos. Nas próximas secções detalhamos estas abordagens, descrevendo os procedimentos e técnicas que podem ser utilizados para a adoção de cada uma delas no desenvolvimento dos casos de teste.

2.2.1 White Box Testing

A abordagem de *White box* testa tendo em conta o funcionamento interno de um sistema ou componente [10]. Esta abordagem é usada para verificar se o código do sistema funciona da forma esperada testando componentes internos do software. Requer um conhecimento do código, conhecimentos de programação e da implementação do software para identificar todos os caminhos que o software pode tomar.

Benefícios:

- Os testes podem ser iniciados numa fase inicial do ciclo de desenvolvimento.
- Os testes acabam por ser mais completos, devido ao facto de poderem cobrir mais caminhos, já que é conhecido o funcionamento interno sendo mais facil criar casos testes para cobrir caminhos específicos.

2.2.2 Black Box Testing

A técnica de *black box* tem como objectivo ser completamente indiferente ao comportamento interno e estrutura do programa. Assume uma perspectiva externa e foca-se no utilizador final, verificando que se aquilo que estará acessível ao utilizador funciona de forma correcta e se vai de encontro aos requisitos e especificações. Assume-se que não existe um conhecimento do código e do funcionamento interno do sistema.

Estes testes podem ser funcionais ou não funcionais. Essencialmente neste tipo de testes o engenheiro de testes, seleciona entradas válidas e inválidas e determinada se o resultado é o esperado.

Benefícios:

- Testes realizados do ponto de vista do utilizador final.
- Ajuda a detectar possíveis ambiguidades e inconsistências entre o sistema e as especificações.
- O engenheiro de testes não precisa de possuir um conhecimento detalhado do sistema.

2.3 Testes manuais e automáticos

Existem duas formas de criar testes, de forma manual ou automática. O teste manual é o processo através do qual os engenheiros de teste executam testes manualmente, comparando os resultados esperados do software com os resultados reais. Podem ser realizados por *testers* mas também por utilizadores reais, de forma, a realizar acções típicas, para encontrar erros. Em ambas as abordagens não existe garantias que todas as funcionalidades serão cobertas.

Como já referido no capítulo 1, com a complexidade actual das interfaces gráficas o processo manual de testes torna-se muito caro, demorado e executado poucas vezes. Por isso, é necessário arranjar soluções que permitam automatizar este processo. Existem várias técnicas para automatizar os testes, o teste baseado em modelos é uma delas. Nesta técnica o modelo do sistema é usado para a geração de casos de teste. Esta técnica será aprofundada no capítulo ??.

Algumas vantagens da automação:

- **Confiável** - o teste executa sempre a mesma operação cada vez que executa, eliminando possíveis erros humanos
- **Repetitivo** - o teste pode ser executado as vezes que forem precisas
- **Reutilizáveis** - os testes podem ser reutilizados em versões diferentes da aplicação
- **Mais qualidade** - pode-se executar mais testes em menos tempo com menos recursos
- **Mais rápidos**- executar testes automáticos é significativamente mais rápido do que testes manuais.

2.4 Avaliação de interfaces gráficas

Com a crescente complexidade das GUI, é cada vez mais uma tarefa difícil e desafiante testá-las. As GUI estão presentes em muitos sistemas de software. Existe um grande número de combinações de entrada e eventos que podem ocorrer, criando assim um grande número de estados diferentes. O correcto funcionamento da GUI é essencial para garantir o bom funcionamento do sistema de software. É possível encontrar três tipos de erros numa GUI [11]:

- **Funcionais** - encontram-se relacionadas com funcionalidades do sistema.
- **Erros de desempenho** - estão relacionados com erros não funcionais, como por exemplo, a eficiência do sistema.

- **Usabilidade** - estão relacionados com a dificuldade do utilizador de utilizar o sistema e as suas funcionalidades.

2.4.1 Testes Funcionais

Os testes funcionais focam-se na funcionalidades do sistema, i.e, no comportamento da aplicação. São derivados a partir da especificação do sistema ou do componente a ser testado. Quando se realiza um teste funcional testa-se aquilo que se espera que o sistema vá fazer incluindo entrada de dados, execução e resposta.

O teste funcional pode ser aplicado em todas as fases de teste (unitário, integração, sistema e aceitação).

2.4.2 Testes de Usabilidade

A usabilidade mede em que medida os utilizadores conseguem utilizar um sistema com satisfação, eficácia e eficiência na realização de tarefas. Este tipo de teste tem como objectivo verificar a facilidade com os utilizadores compreendem e interagem com a interface gráfica. Existem várias tecnicas para analisar a usabilidade de uma interface gráfica, como por exemplo, os métodos de inspecção e baseados em inquéritos.

Métodos de inspecção

São um conjunto de métodos baseados na opinião de peritos que inspecionam ou examinam aspectos de usabilidade de uma interface. São exemplo as técnicas, Cognitive Walkthrough [12] e avaliação heurística [13].

Na *avaliação heurística*, o especialista avalia a usabilidade do sistema comparando o mesmo com uma série de critérios definidos a priori, que são considerados boas práticas, como por exemplo, falar na linguagem do utilizador, prevenir erros, fornecer feedback e ser consistente.

Cognitive Walkthrough é um método de avaliação de usabilidade, em que um ou mais avaliadores realizam uma série de tarefas, que um utilizador típico terá de realizar. Em cada tarefa, identificam quais os problemas ou dificuldades que os utilizadores teriam ao realizá-la, de acordo com os objetivos e conhecimento dos mesmos. Ao contrário da avaliação heurística, existe um processo bem definido para realizar o *Cognitive Walkthrough*. Em vez de avaliar a conformidade das "regras", os avaliadores observam directamente os utilizadores a realizar as tarefas definidas.

Inquérito

Nas técnicas baseadas em inquéritos, o avaliador observar os utilizadores a interagirem com o sistema a tentar completar tarefas reais que irá realizar no dia a dia, ao invés de tarefas definidas pelo avaliador como acontecia nos métodos de inspecção.

Nesta técnicas os utilizadores experimentam o sistema e ao mesmo tempo respondem a um inquérito sobre a sua experiência com a interface gráfica. Assim é possível obter informações sobre os gostos dos utilizadores, necessidades e compreensão do sistema conversando com eles e observando-os a usar o sistema em tarefas reais.

2.5 Construção e execução de testes automáticos para Web

Existem várias ferramentas que permitem realizar testes de forma automática, sendo possível a partir delas simular a navegação num *browser* da mesma forma que um utilizador, clicar em *links*, preencher formulários, carregar em botões e podendo verificar se o resultado dessas ações é o esperado. Apesar de permitirem a realização dos testes de uma forma não assistida, é ainda necessário o criar os casos de testes e fazer o mapeamento para a GUI para que seja possível interagir com ela.

Existem várias ferramentas disponíveis, sendo as mais populares o WatiR e o Selenium.

WatiR é uma ferramenta *open source* de testes para aplicações Web. WatiR é construída em Ruby e o seus principais objetivos são testar sistemas de larga escala, realizar testes funcionais e automatizar testes de aceitação do utilizador. Esta ferramenta permite um controlo directo sobre o DOM numa página web [14]. As principais funcionalidades do WatiR incluem o suporte a diversos browsers, suporte para eventos Javascript, *frames* e campos escondidos, a possibilidade de esperar até que a página esteja completamente carregada antes de continuar a interacção com a página e a captura de imagens do ecrã.

No exemplo a seguir será demonstrado um exemplo de uma pesquisa no google com recurso ao *WatiR*. A Figura 2.1 apresenta:

- **Passo 1:** Abrir Browser
- **Passo 2:** Ir para www.google.com
- **Passo 3:** Introduzir ”o que é o watir?” na search box
- **Passo 4:** Clicar no botão de submit
- **Passo 5:** Comparar o resultado actual com o resultado esperado

```
class GoogleSearch < Test::Unit
def test_search
@browser.goto "www.google.com"              #PASSO 1
@browser.text_field(:name => "q").set "o que é o watir?"  #PASSO 2
@browser.button.clique                      #PASSO 3
@browser.div(:id => "resultStats").wait_until_present   #PASSO 4
assert @browser.title == "o que é o watir? - Google Search" #PASSO 5
end
end
```

Listagem 2.1: Exemplo de ficheiro de testes em WatiR(adaptado de [14])

Selenium é uma outra ferramenta para testar aplicações web pelo browser de forma automatizada. Fornece um ambiente de desenvolvimento integrado (Selenium IDE). O Selenium IDE pode ser usado para gravar, editar e depurar *scripts* de teste. Os *scripts* de teste são escritos em Selenese, a linguagem de script de teste usado pela Selenium. Esta linguagem fornece comandos para as acções realizadas nos navegadores (como o clique em um *link* ou a selecção de uma opção) e também para a recuperação de dados a partir das páginas resultantes. O Selenium é capaz de escrever testes numa série de linguagens de programação, incluindo C#, Java, Groovy, Perl, PHP, Python e Ruby. Sendo esta ferramenta semelhante ao WatiR a nível de funcionalidades [15]. Na Figura 1 é apresentado um pequeno exemplo com instruções Selenium, para automatizar um *login*. O Selenium possui as mesmas funcionalidades que o WatiR.

```
public class LoginPage {
    private final WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    public void loginAs(String username, String password) {
        driver.get("http://url_to_my_webapp");
        driver.findElement(By.id("username")).sendKeys("demo");
        driver.findElement(By.id("pwd")).sendKeys("demo");
        driver.findElement(By.className("button")).submit();
    }

    public static void main(String[] args){
        LoginPage login = new LoginPage(new FirefoxDriver());
        login.loginAs("user", "pass");
    }
}
```

Listagem 1: Exemplo de ficheiro com instruções Selenium

Uma outra ferramenta capaz de automatizar os testes sobre interfaces gráficas é o Sikuli. É uma tecnologia nascida no MIT UI Group Design, que permite automatizar as operações do computador usando visão

computacional. Esta ferramenta usa uma abordagem diferente da usada pelas ferramentas anteriores. Usa reconhecimento de imagem para identificar e controlar os componentes da GUI. Pode ser útil quando não há um acesso fácil aos elementos de uma GUI. Utiliza algoritmos de visão computacional para analisar a interface gráfica [16].

2.6 Conclusão

Neste capítulo foi feita uma introdução ao tema de testes para melhor compreender a temática principal da dissertação e discutida qual a sua importância no ciclo de desenvolvimento de software. Foi apresentado o processo de testes de um sistema de software, os diferentes níveis e tipos de testes. Foram também apresentadas algumas ferramentas existentes que permitem a realização de testes de forma automática. Entender o processo de teste de software é essencial para este projeto, uma vez que um dos objetivos principais deste trabalho é de criar uma ferramenta capaz de gerar e executar testes sobre interfaces gráficas.

Existem várias abordagem para testar a interface gráfica com o utilizador[[11] [4] [17]. Existem técnicas de teste para avaliar a usabilidade, que se focam na opinião e nas dificuldades dos utilizadores. Um outro conjunto de técnicas foca-se na implementação, e procuram garantir a qualidade do sistema, quer em termos funcionais, quer de desempenho. É importante notar que, para que um sistema tenha boa usabilidade, é importante garantir o seu bom funcionamento.

Nesta dissertação irão ser abordadas técnicas focadas na qualidade da implementação. Mais concretamente, as técnicas de testes baseadas em modelos. Os testes baseados em modelos só encontram erros relacionados com desvios do que está dito no modelo. Assim, analisar usabilidade é complicado. O principal objectivo é encontrar erros de funcionalidades ou falhas. No entanto, permitem reduzir os altos custos dos testes sobre interfaces gráficas.

3 | Testes baseados em modelos

Modelar na engenharia de software é o processo de criação de uma representação de um sistema, abstraindo e simplificando o seu comportamento ou estrutura, ou seja, é uma forma de representar a estrutura/comportamento de um sistema [18]. Os modelos são mais simples do que o sistema que descrevem e por isso ajudam a mais facilmente o entender.

MBT (Model-Based Testing) refere-se a um processo de engenharia de software que estuda, constrói, analisa e aplica modelos bem definidos para dar suporte nas varias atividades relacionadas com os testes. É uma técnica de *black-box* que tem por objectivo verificar se a implementação de um software se encontra de acordo com a sua especificação/modelo e foca-se na geração automática de testes.

A ideia básica é identificar e construir um modelo abstracto que represente o comportamento do SUT (*System Under Test* – Sistema sob teste). Com esse modelo é possível gerar um grande número de casos de teste. No MBT o modelo serve ainda de oráculo dos testes, ou seja, quando os testes são executados no sistema, o resultado é comparado com o que é dito no modelo. É possível gerar casos de testes de duas formas distintas no MBT, *online* e *offline*. Em modo *offline* os casos de teste são gerados antes de serem executados. No modo *online* os casos de teste são gerados enquanto os testes estão a ser executados.

3.1 Processo do model-based testing

O processo de MBT (Figura 3.1) pode ser dividido em cinco passos [2]:

1. **Modelar o SUT**
2. **Gerar os testes abstractos a partir do modelo**
3. **Transformar os testes abstractos em testes executáveis**
4. **Executar os testes no SUT**
5. **Analisar os resultados**

O primeiro passo é construir o modelo abstracto do sistema. Deve ser focado nos aspectos chave daquilo que será testado, omitindo detalhes irrelevantes. O modelo deverá estar de acordo com requisitos e deverá conter o comportamento esperado do SUT. De seguida, serão gerados os testes abstractos a partir do modelo, usando algum tipo de critério (por exemplo, escolher um critério em que os testes devem cobrir todas as transições). O resultado desta fase será uma sequência de operações. Depois os testes abstractos serão transformados em testes concretos e executáveis. Uma das vantagens de possuir estas duas camadas, é que os testes abstractos podem ser independentes da linguagem de implementação utilizada na escrita dos testes concretos que serão executados no SUT. Deste modo é possível reutilizar os testes abstractos, se a linguagem de implementação for alterada. Na fase seguinte são executados os testes. No final é realizada a análise dos resultados, verificando se os mesmos se encontram consistentes com os resultados esperados.

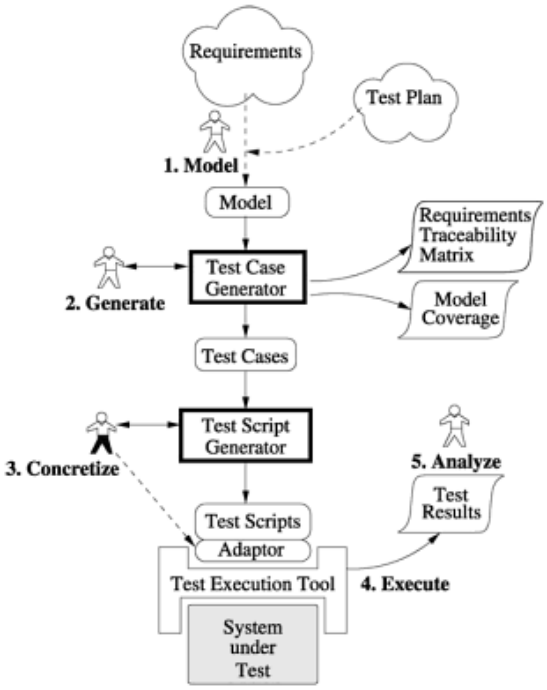

Figura 3.1: Processo de Model based Testing (adaptado de [2])

3.1.1 Vantagens

Hoje em dia a mudança de requisitos e especificações é uma constante e é inevitável mesmo quando o processo de desenvolvimento já está iniciado. As interfaces gráficas estão sujeitas a alterações constantes ao longo do ciclo de vida de um produto software. É por isso importante minimizar os custos dessas mudanças. Quando as especificações mudam, apenas algumas partes relevantes do modelo é que precisam

de ser alteradas. E como a partir do modelo é possível gerar os testes de forma automática, basta gerá-los de novo, aproximando o custo de manutenção a zero [19]. Uma vez que a fase de geração de testes é automatizada, o maior custo vai para o desenvolvimento do modelo e para o mapeamento entre o modelo e a implementação [20]. Seguem-se alguns benefícios do MBT segundo [11]:

- A construção do modelo de comportamento pode ser iniciada no principio do ciclo de desenvolvimento.
- Usando esta técnica é possível aumentar a qualidade do software, enquanto que são reduzidos outros factores como tempo e esforço gasto em escrever testes e analise de resultados.
- A modelação expõe ambiguidades na especificação do sistema, levando a corrigir código desde o início.
- Reutilização dos modelos, que são mais facilmente adaptados e passíveis de sofrer alterações. Os modelos podem ser reutilizáveis no futuro mesmo quando as especificações mudam.
- Maior grau de automação, permitindo realizar testes de forma mais exaustiva.
- Confiabilidade: os testes realizam sempre a mesma operação cada vez que correm, eliminado erros humanos.
- Evolução de requisitos - com a necessidade de introduzir ou actualizar requisitos basta atualizar modelo e gerar automaticamente novos testes.
- Número essencialmente ilimitado de diferentes testes que podem ser gerados, algo que um engenheiro de testes não conseguiria por ele realizar de forma manual [2].
- Quantidade reduzida de manutenção de teste (manutenção de modelos em vez de grandes conjuntos de scripts de teste)
- MBT permite aos engenheiros de testes envolverem-se mais cedo no ciclo de desenvolvimento [19].

3.1.2 Limitações

O processo de MBT não oferece apenas vantagens. Um dos principais problemas é o facto de o modelo poder estar diferente do SUT. São ainda precisas competências para criar um modelo abstracto. Algumas das suas principais limitações do MBT estão descritas abaixo.

- Requer um modelo - Ficando depende sempre do que é possível expressar com o modelo.

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

3.2 Model-based testing em interfaces gráficas

Ao testar a interface gráfica de um software é possível detectar não só erros da interface mas também erros relacionados com a aplicação. O MBT pode ser utilizado nos testes da interface gráfica com o utilizador (GUI), de forma a testar se a interface desenvolvida vai de encontro ao modelo especificado.

3.2.1 Trabalhos relacionados

Existem alguns trabalhos de pesquisa na área da automação de casos de testes baseado em modelos para interfaces gráficas. Atif M. Memon foi o primeiro a apresentar um trabalho nesta área [17]. Outras abordagens foram efectuadas nesta área, como é o caso do trabalho desenvolvido por Ana Paiva [11]. Existem mais alguns autores que apresentaram diversas abordagens.

Memon apresentou uma *framework* de testes sobre GUI's o ''GUITAR'' que inclui um método de modelação baseado em eventos. Uma GUI possui um grande número de *widgets* que esperam eventos. A ferramenta é usada para gerar modelos de aplicações Java GUI com fim de executar testes. Primeiro o GUITAR, extrai da aplicação informações sobre a estrutura de todas as janelas, *widgets*, assim como os seus atributos e eventos da interface gráfica e cria um ficheiro XML. A ideia é criar um fluxo de eventos com todas as possíveis interacções de eventos na GUI. Estes são então utilizados para gerar casos de teste de GUI que são sequências de eventos de GUI. A ferramenta GUITAR também suporta a execução dos casos de teste gerados sobre a aplicação Java GUI. Uma das suas desvantagens é não haver suporte para a refinação manualmente dos modelos gerados.

Paiva desenvolveu um *add-on* para o Spec Explorer, uma ferramenta de testes baseada em modelos desenvolvida pela *Miscrosoft*, de forma a adaptá-la para testes sobre interfaces gráfica [11]. No seu trabalho, utiliza métodos de teste baseados em especificações formais de forma a sistematizar e automatizar o processo de teste de interfaces gráficas com o utilizador. A abordagem descrita incorpora modelos gerados a partir da análise do código fonte, servindo de base para uma ferramenta de teste de interfaces gráficas via modelos Spec# e usando a ferramenta Spec Explorer [11].

Outra abordagem pelos mesmos autores é o de Pattern Based GUI Testing (PBGT), que visa promover a reutilização de estratégias de teste para testar comportamentos comuns na web. É baseada numa linguagem PARADIGM, que tem por objectivo simplificar e diminuir o esforço necessário no processo de modelação promovendo a reutilização. Os casos de teste são gerados automaticamente a partir dos modelos PARADIGM.

Cruz e Campos [4], possuem outra abordagem que consiste em gerar uma máquinas de estado a partir de um modelo de tarefas. Inicialmente é necessário criar um modelo de tarefas. Depois usando uma ferramenta chamada TERESA [21] é gerada uma representação chamada de Presentation Task Sets (PTS). A partir desta e do mapeamento do modelo para a interface real é criado um grafo a partir do qual são gerados os casos de teste. Podem ainda ser introduzidas mutações nos casos de teste, para representar enganos por parte de utilizadores no uso da interface gráfica.

3.3 Mutações

Nesta secção irá ser apresentado o conceito de testes positivos e negativos. Irá ser introduzido o conceito de mutação e quais os tipos de mutações existentes para aplicações *Web*. Uma mutação é um método de inserção de falhas em testes de software. É uma técnica para avaliar e melhorar um conjunto de testes.

3.3.1 Testes positivos e negativos

Os testes positivos são testes que seguem o caminho que o utilizador deve seguir, ou seja, são testes em que se introduz uma entrada válida e se espera uma acção de acordo com a especificação. Por exemplo, um teste positivo pode ser carregar num *link* e a aplicação ser redirecionada para a página do *link*.

Os testes positivos são importantes mas precisam de ser complementados com testes negativos de forma a perceber qual o comportamento do sistema quando existe um comportamento não esperado (por exemplo, devido a erro seja ele intencional ou não) por parte do utilizador. Os testes devem tentar fazer o sistema falhar e gerar erros de forma a testar o tratamento e recuperação de erros, o comportamento do utilizador não intencional e limites. Os testes negativos podem ter dois resultados:

- se a aplicação funciona como esperado, irá ser recebida uma mensagem de erro e o teste irá passar.
- se a aplicação aceitar a entrada não válida e o teste negativo falha, significa que pode existir um *bug*.

3.3.2 Mutações

As mutações de casos de teste podem ser vistas como testes negativos. Nas mutação de um caso de teste, o teste é alterado de forma controlada e de seguida é executado. A ideia é introduzir pequenos erros para perceber qual o comportamento da aplicação. O objectivo destas mutações é correr os testes de uma forma 'defeituosa' e causar a falha do teste, aumentando assim a confiança e qualidade do software. Na abordagem proposta neste trabalho optou-se por introduzir estas mutações nos caminhos em vez de ser no modelo, de forma a simplificar o modelo ao máximo. Podem existir 3 tipos de comportamentos expectáveis em relação aos testes:

- A expectativa é de que a maioria dos casos de teste mutados deve ser rejeitada já que eles vão produzir casos de teste inválidos logo não poderão ser executados com sucesso.
- Algumas mutações vão passar o processo de validação e produzir um teste válido. Irão produzir resultados diferentes em relação aos casos de teste originais.
- Espera-se que algumas mutações produzam resultados idênticos em comparação com o caso de teste original ou até um comportamento diferente mas que não seja distinguido ou observado.

3.3.3 Tipos de Mutações implementadas

Depois de realizada uma analise foram encontrados três tipos de erros base (slip, lapse e mistake) [22]. No entanto foi decidido ir além desses tipos de erro base e considerar erros mais específicos que podem fazer sentido numa interface web. As mutações foram divididas em dois grandes grupos: mutações em formulários e mutações em cliques em botões ou *links*. As mutações em formulários são as seguintes:

- **Slip** - é realizada uma troca de na ordem de execução de tarefas.
- **Lapse** - é realizada a eliminação de um das acções.
- **Mistake** - modificam valores de input (por exemplo acrescentando mais uns caracteres a uma string válida) podendo ou não tornar o valor errado. São geradas apenas nos métodos que recebem parâmetros de entrada.

- **Múltiplos cliques no botão de submissão** - Por vezes os utilizadores quando o tempo de resposta da página é maior têm tendência a carregar várias vezes no botão de submissão um formulário.

Outro tipo de erro comum por parte dos utilizadores é de realizar duplo cliques em vez de um simples clique. Foram identificados as seguintes mutações possíveis relacionados com este tipo de erro:

- **Duplo clique em elementos** num *link* ou botão em vez de um simples clique.
- **Duplo clique em menus** por parte do utilizador em vez de realizar apenas um clique.

Por vezes, os utilizadores tem tendência a utilizar o back button ou o refresh quando a ligação está mais lenta ou se a página demora um pouco mais a carregar completamente. Torna-se importante saber qual a reacção da aplicação vai tomar quando este tipo de evento (usado muita vezes pelos utilizadores) é disparado. E por isso forma introduzidas estas duas mutações:

- **Carregar no botão back** do *browser*
- **Fazer um refresh à página**

3.4 Conclusão

Neste capítulo foi mostrado como o MBT pode contribuir para a automatização da geração de casos de testes.

O model-based testing é apresentado como um solução para os problemas de automatização existentes na geração de casos de teste. Quando a interface muda, os testes precisam de ser actualizados, para garantir o seu correcto funcionamento. Uma das grandes vantagens do MBT é precisamente quando existe necessidade de mudança, basta actualizar o modelo e gerar de forma automática novos casos de testes. Ainda assim, o MBT requer a criação de um modelo de forma manual. Foram também analisadas as várias abordagens e diversas ferramentas existentes para o MBT.

A grande diferenciação é que em vez de se escreverem os testes manualmente com base nos requisitos e especificações do sistema, é criado um modelo que representa o comportamento esperado do SUT capturando os requisitos e especificações. A partir desse modelo são gerados os casos de teste de forma automáticas. Foi também introduzido o conceito de mutação e a sua importância nos testes de software.

4 | Modelação de comportamento

Na fase de design e concepção de uma interface gráfica, são gerados mockups e mapas de navegação. Os primeiros representam os elementos presentes em cada página. Os segundos, o diálogo que é possível estabelecer com a aplicação. Existem muitas abordagens para modelar uma interface gráfica [2]. Assim, na abordagem proposta nesta dissertação optou-se por representar a interface gráfica como uma máquina de estados. No caso de interfaces gráficas para aplicações web uma página pode ser representada como um estado que pode ter vários sub-estados. Cada link ou botão dessa página representa acções de *input* ou comandos que são transições que levam a outros estados.

4.1 Máquinas de estado

Máquinas de estado são um padrão recorrente na engenharia de software. São vistas como uma maneira útil de pensar sobre o comportamento de sistemas reactivos, desde as fases iniciais de desenho, até aos testes de software [23]. Quando o sistema recebe *inputs*, ele executa algum tipo de ação podendo mudar o seu estado, onde receberá novamente *inputs*. Os sistemas que correspondam a esta é são conhecidos como sistemas reativos e, por exemplo, interfaces gráficas do utilizador encaixam-se dentro desta categoria. Máquinas de estado são modelos adequados de tais sistemas [11] [18].

Uma máquina de estados é um modelo que possui um conjunto de estados, bem como transições para outros estados, que são acionadas por eventos externos (originados por exemplo, por utilizadores) ou até por eventos do próprio sistema.

$$M = (S, A, t)$$

- Os elementos em S são chamadas de **estados**.
- Existe um elemento de S distinto chamado de **estado inicial**.
- O conjunto A é chamado o alfabeto de entrada

- A função t é chamada de função de transição de estado que representa os eventos. Em que

$$t : S \times A \rightarrow S$$

Em qualquer momento o sistema é suposto estar nalgum estado $s \in S$. A máquina lê um *input* $a \in A$ e faz a transição para o estado $t(s, a)$.

4.2 Utilização no âmbito dos testes

Uma FSM pode ser vista como um grafo orientado, em que os testes podem ser gerados usando algoritmos para travessia sobre grafos [24]. Um modelo FSM é representado como um grafo composto por um número (finito) limitado de nós (estados) ligados através de arestas dirigidas (transições). Assim testar uma aplicação é semelhante a atravessar um caminho através de um grafo, utilizando técnicas derivadas da teoria dos grafos, o que permite de forma fácil e eficiente usar a informação comportamental do modelo para gerar sequências novas e úteis de teste [25]. Os caminhos podem ser restringidos, por exemplo, forçando o teste a começar e terminar no mesmo estado inicial definido ou limitando o número de estados que o caminho deve visitar, dependendo das necessidades de teste.

Uma FSM pode ser visualizada sendo possível ver quais os caminhos que os testes podem tomar, já que possuem uma notação gráfica, tornando a sua compreensão mais fácil. Fornecendo assim um mecanismo fundamental para testar o comportamento de um software sem ser necessário considerar qual a tecnologia em que ela será implementada.

4.3 SCXML

Uma FSM pode ser representada em SCXML (State Chart XML). O State Chart XML permite descrever um modelo de máquinas de estados num dialecto XML. Possui eventos, transições e outros marcadores para descrever o comportamento de uma máquina de estados. É possível descrever máquinas de estado complexas usando esta linguagem baseada em XML [26]. A linguagem XML foi utilizada não apenas por apresentar os dados de uma maneira estruturada, mas também por ser flexível, permitindo que a estrutura a ser utilizada seja definida pelo programador. O SCXML é baseado em Harel Statecharts [27], que também só a base dos diagramas de estados do UML [28]. Harel Statecharts são uma representação matemática das máquinas de estado. SCXML é actualmente um *working draft* publicado pelo W3C [29]. Na Listagem 2 é apresentado uma máquina de estado em SCXML e na Figura 4.1 a sua representação visual. O SCXML possui a seguinte semântica:

- **<state>** - representa um estado. Pode conter vários atributos e podem ter ou não subestados e cada um destes estados pode ter zero ou mais transições.
- **<transition>** - As transições entre estados só accionadas por eventos e podem ser condicionais. O atributo "target" especifica o destino da transição que pode ser um <state>.

```
<?xml version="1.0" encoding='us-ascii'>
<scxml version='1.0"  initialstate='S1'>
  <state id="S1'>
    <transition event="Event1" target="S2"/>
  </state>

  <state id='S2'>
    <transition event="Event2" cond="X>0" target="S1"/>
    <transition event="Event2" cond ="X<0" target next="S3"/>
  </state>

  <state id='S3'>
  </state>

</scxml>
```

Listagem 2: Exemplo de representação em SCXML

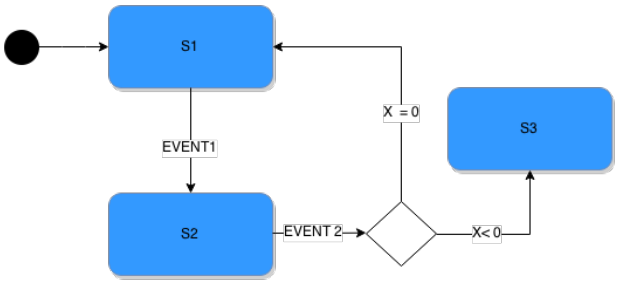

Figura 4.1: Representação gráfica de uma máquina de estados

4.3.1 SCXML como modelo de interfaces gráficas

Cada página é representada por uma tag **<state>** e que o atributo **id** da tag é utilizado para identificar a janela. Cada um dos elementos descritos de seguida que tiverem atributos como *id* ou *label*, servirão para mapear com os ficheiros de configuração.

Dentro da tag **<state>** podem existir tags **<transition>**, representando cada uma das possíveis transições possíveis na página, por exemplo um clique num link ou botão que origine uma transição de estado.

A tag **<transition>** tem os seguintes atributos:

- um *ID*
- um *target* que será o *id* do estado para onde irá transitar.

Existe a tag **<step>** dentro da tag **<transition>** já que pode ser necessário mais de que uma acção para passar a outro estado, por exemplo, se o clique for num menu em que é necessário passar o rato por cima e depois clicar no link. Possui:

- um atributo *ID*

Dentro da tag **<state>** podem existir **<onentry>**, que representam as validações que serão feitas sempre que se entrar num certo estado.

- um atributo *ID*
- um atributo *type* - que pode ser $displayed?$,$disabled?$ por exemplo. Estes tipo serão explicados mais a frente.

A tag **<state>**, poderá ainda conter outros sub-estados (que poderão representar um formulário por exemplo), que serão representado também por tags **<state>**.

- um atributo *ID*
- um atributo *type* para identificar o tipo (pode ser estado ou sub-estado)

Finalmente a tag **<onexit>** representam as validações que serão feitas sempre que se sai de um certo estado, os seus atributos são:

- um atributo *ID*
- um atributo *type* - que pode ser $displayed?$,$disabled?$ por exemplo. Estes tipo serão explicados mais a frente.

Na Listagem 3 é apresentado um exemplo de um modelo representado em SCXML.

```
<scxml xmlns="http://www.w3.org/2005/07/scxml" initial="X'>
 <state id="X'>
  <!-- STATE ELEMENTS -->
  <state id="X_1" type="Form" >
   <!-- SEND ELEMENTS -->
   <send label="A" type="A"/>
   <send label="B" type="A"/>
   ....
   <!-- TRANSITION OF SUB STATE -->
```

```
    <transition type="X" label="X_1_submit" >
      <submit target="Y" />
      <error  target="E" />
    </transition>
  </state>
  ....
  <!-- TRANSITION UNIQUE ACTION ELEMENTS -->
  <transition id="Z" target="Z" />
  ...
  <!-- TRANSITIONS WITH MULTIPLE ACTIONS ELEMENTS -->
  <transition id="W" target="W" type="menu'>
    <step  id="W1" target="W" />
    <step   id="W2" target="W" />
    ...
    ...
  </transition>
  ...
  <!-- ONENTRY ELEMENTS -->
  <onentry id="A" type="B"/>
  <onentry id="B" type="C"/>
  ...
  ...
 </state>
 ....
</scxml>
```

Listagem 3: Formato do SCXML

4.4 Exemplo de modelação de uma interface gráfica em SCXML

O modelo em SCXML deverá conter a informação referente à interface que queremos testar. O SCXML é a linguagem usada para representar uma máquina de estados, no entanto, foi necessário introduzir outros elementos para poder representar todas as funcionalidades presentes numa interface gráfica. Assim é utilizado um esquema customizado. Como é baseado em XML, o modelo é assim compreendido e editado por humanos, por máquinas e pretende ser o mais abstracto possível. O modelo terá então de seguir certas regras para que o modelo seja validado.

4.4.1 Representação em SCXML

A representação da interface gráfica será constituída por um elemento raiz que é um *state*. Cada um dos estados poderá ter:

- Um ou mais sub-estados (que poderão representar um formulário por exemplo), representado por *<state>*

- Uma ou mais transições que irão representar *call's* na interface(e.g *clicks, links*), representado por *<transition>*

- Uma ou mais validações que representaram testes que cada vez que se entrar nessa página serão realizados, representado por *<onentry>* e *<onexit>*

Cada sub-estado possui:

- *input*s de utilizador(tipicamente *text boxes, check boxes*, etc)

- Transições para outros estados

- Transições para estado de erros

- Validações a serem feitas depois de sair dele ou ao entrar nele.

Cada transição possui:

- uma ou mais acções sequências necessárias para passar a outro estado (por exemplo,para carregar num *link* que se encontra num menu é preciso carregar no menu e depois carregar no *link*).

- zero ou mais elementos *<send>* que representa *input*s por parte do utilizador.

4.4.2 Exemplos

Serão agora apresentados pequenos e simples exemplos, de forma a ilustrar como se pode modelar um interface gráfica com SCXML.

Na Figura 4.2 é representada uma maquina de estados de uma interface gráfica. Na Listagem 4 é apresentado o SCXML para a maquina de estados da Figura 4.2. O primeiro estado é a pagina de entrada da aplicação. Dentro dessa página existe um formulário de *login* e um *link* para a página de registo. O formulário de *login* irá ser representado como um subestado e o *link* como uma transição para outro estado. Quando o formulário de *login* é preenchido com sucesso, há uma transição para outro estado.

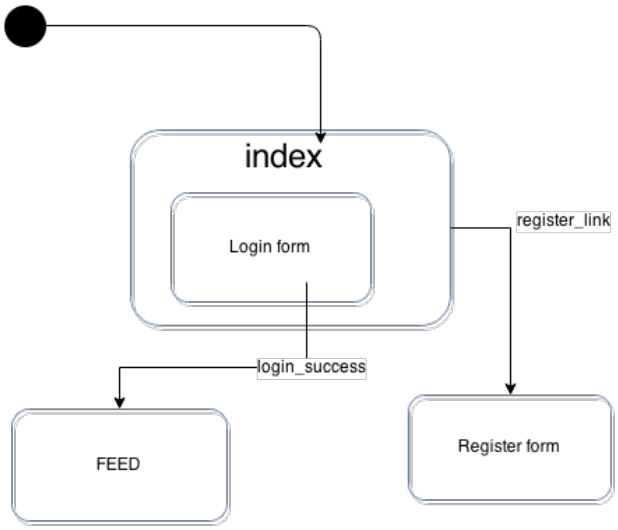

Figura 4.2: Máquina de estados

```
<state id='index'>

  <state id="login\_form" type="form" >
    <send label="email" type="required"/>
    <send label="password" type="required"/>

    <transition  type="form" label="submit_login\_form" >
      <submit target="feed" />
      <error target="login_error" />
    </transition>

    <!--validation-->
    <onexit id="message_authentication" type="default"/>

  </state><!-- END OF LOGIN FORM -->

  <!-- LINKS -->
  <transition id="register" target="register"/>

  <!-- Validations onentry in state -->
  <onentry id="navbarAgro" type="css"/>

   <!-- SUBESTADO 2 -->
  <state id="feed'>
     <!-- LINKS -->
     <transition id="mercado" target="mercado" type="menu>
       <step  id="message_hover" target="mercado" />
       <step  id="menu_mercado" target="mercado" />
       <step   id="mercado" target="mercado" />
     </transition>
```

```
    <transition id="feed" target="feed" />

    <!-- LINKS -->
    <transition id="perfil" target="perfil" />

    <onentry id="publicidade" type="displayed?"/>
    <onentry id="destaques" type="displayed?"/>
  </state>

</state><!-- END OF INDEX PAGE-->
```

Listagem 4: Exemplo de modelo em SCXML

Na Listagem 4 temos o caso de uma página que possui dois sub-estados. Na primeira página existe um sub-estado que é um formulário de *login*. Neste sub-estado com o *id* **login_form**, é possível ver que possui elementos *<send>* que representam o *textbox* para o utilizador introduzir o seu *email* e a sua *password*. Este estado **login_form** possui uma transação . Quando o formulário será submetido com sucesso ele irá para um estado **feed**, quando o formulário terá um erro será redirecionado para um estado **login_error**. Ainda no sub-estado **login_form** é possível ver que existe uma validação do tipo *default* que é feita sempre sair dele, por exemplo, sempre que o formulário de *login* for submetido terá de ser feita uma validação.

Depois na outra página representada pelo estado **feed**, que possui transições para os estados **perfil**,**feed**,**mercado**, estas transições podem representar na página simples *links* simples ou com mais ações como por exemplo integrações com menus como é o caso da transição com *id* **mercado** para outras páginas. Possui ainda validações de entrada no estado, neste caso concreto possui duas validações do tipo **displayed?** que verificam se os elementos aparecem na página.Com a descrição acima feita é possível perceber qual a relação concreta entre as páginas web e o modelo que representa essas páginas.

O proximo exemplo será de um formulário numa página de pesquisa de voos. A página da aplicação de pesquisa de voos presente na parte superior da Figura 4.3 representa o estado **flights** no modelo presente na parte inferior da Figura 4.3. Depois é possível ver que existe um sub-estado **flights_form** que representa o formulário de pesquisa de voos. Neste sub-estado os elementos **<send>** representam os vários campos do formulário. Cada um destes campos tem um atributo de **type** que pode ser obrigatório ou opcional e pode ainda ter um atributo a referir qual o tipo de elemento se é **select box**, **check box**.

O botão de submissão do formulário é representado na tag de **transition**. E esta transição diz que se o formulário for submetido com sucesso então a aplicação para o estado **flights_results**, se não for submetido com sucesso então irá para o estado **flights_form_error**.

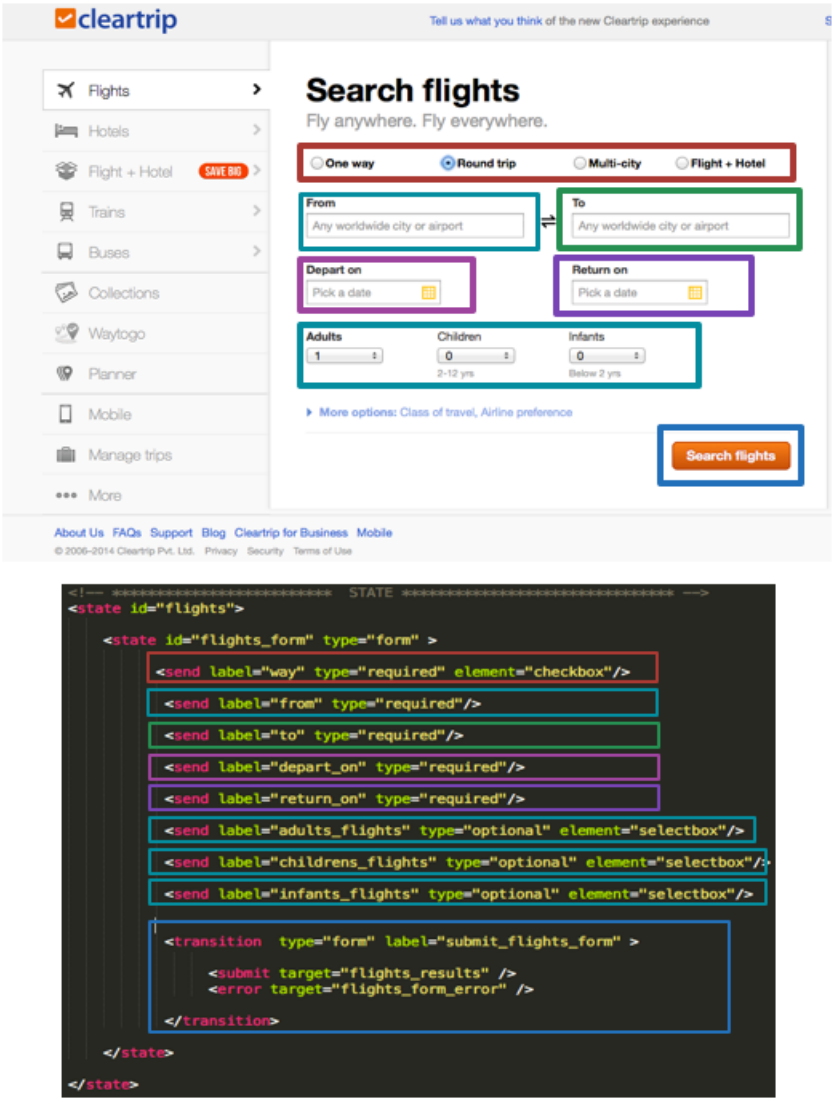

Figura 4.3: Mapeamento entre implementação e modelo

4.5 Conclusão

Neste capítulo, foi apresentado uma forma de modelar o comportamento de uma interface gráfica usando máquinas de estado. Ao usar uma FSM, a ideia é que o engenheiro de testes represente o comportamento da GUI como se fosse uma máquina de estados onde cada *input*/evento/acção realize uma transição de estado. Para representar a máquina de estados pode ser usado o SCXML, que possui elementos para descrever o comportamento de uma FSM.

Conclui-se que o comportamento de uma GUI pode ser modelada usando FSMs e assim casos de teste podem ser gerados automaticamente percorrendo os caminhos através do modelo da aplicação, em que cada caminho representa um caso de teste.

5 | Uma abordagem para testes baseados em modelos

5.1 Abordagem proposta

A abordagem proposta consiste em utilizar um modelo da interface gráfica para gerar os casos de testes para essa mesma interface gráfica. Assim o **processo de MBT** proposto segue os seguintes passos:

1. Criação do modelo da interface gráfica.
2. Definir o mapeamento entre o modelo e a interface concreta.
3. Gerar um grafo a partir do modelo.
4. Aplicar algoritmos de travessia sobre o grafo e gerar sequência de testes abstractos.
5. Gerar testes executáveis.
6. Executar os testes e verificar o seu resultado.

Neste processo apenas o passo 1, 2, 6 são realizados de forma manual, os restantes são todos automatizados. A Figura 5.1 mostra a arquitectura do processo para gerar e executar os casos de testes.

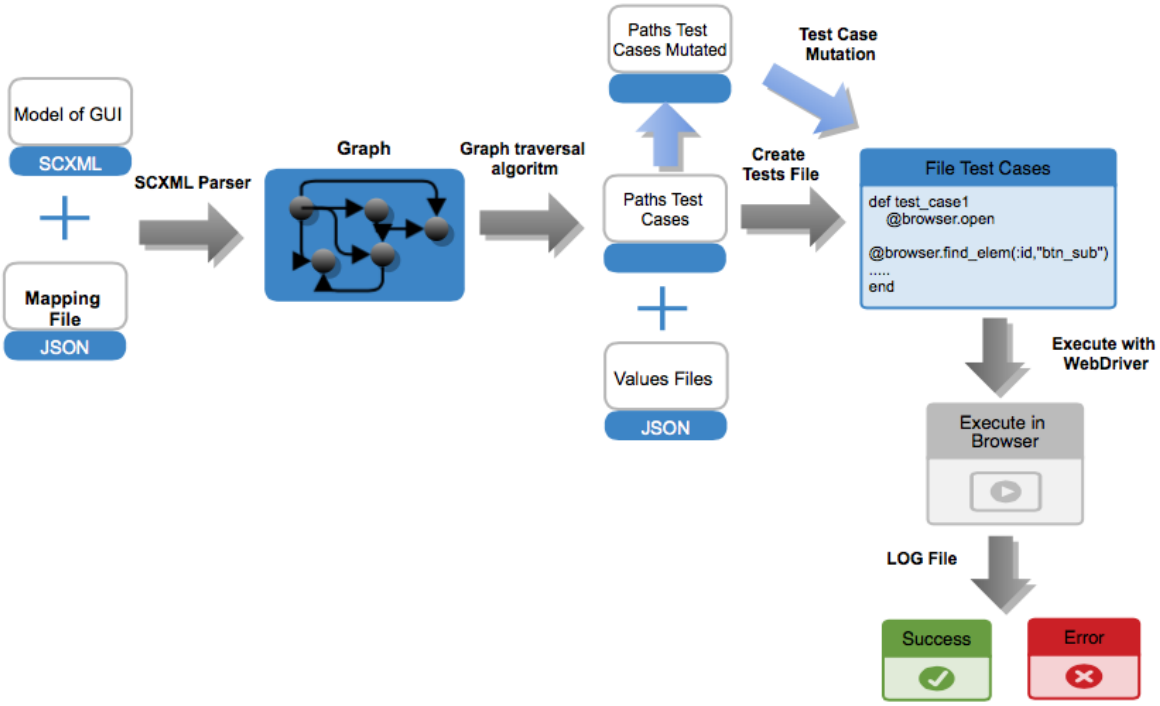

Figura 5.1: Arquitectura do processo de testes

Numa primeira fase será necessário criar o modelo que representa o comportamento da interface gráfica. Ao ser baseado num modelo representado por uma máquina de estados. A ideia é que os testes consigam fazer com que a aplicação realize as transições necessárias entre os vários estados da aplicação em que as transições sejam disparadas por eventos. Em que estes eventos representem o comportamento esperado dos utilizadores no uso da interface gráfica. Em conjunto com este modelo será necessário definir o mapeamento entre o modelo e a interface gráfica, bem como os valores de entrada a utilizar para a geração de casos de testes.

O modelo que será uma máquina de estados, que será transformada num grafo. A partir daqui será necessario percorrer esse grafo de forma a gerar caminhos. A partir desses caminhos gerar o código dos testes executáveis. Os casos de teste serão gerados em Java e serão executados com recurso à *framework* Selenium. No final serão executados e gerados relatórios com detalhes da execução dos testes.

Por vezes, os utilizadores podem ter comportamentos inesperados e fora daquilo que é esperado, por isso serão introduzidos pequenos erros nos casos de teste, para simular esses tais comportamentos.

5.2 Modelo

A abordagem para representar o comportamento de uma interface gráfica é apresentado no Capitulo 4.

5.3 Ficheiros de Configuração

Uma vez completa a criação do modelo, é necessário saber a que elementos da aplicação web a testar correspondem os elementos do modelo. O ficheiro de mapeamento das variáveis do modelo contem para cada uma das variáveis do modelo qual o tipo de elemento HTML a encontrar, como encontrá-lo e qual o tipo de ação a ser executada. Este ficheiro deve conter todas as variáveis que existem no modelo. É necessário criar dois ficheiros de configuração para o processo de geração de testes:

- Um ficheiro de mapeamento entre os elementos do modelo e os da interface gráfica a testar.
- Um ficheiro com valores e dados para os testes.

O ficheiro de mapeamento apresentado na Listagem 5 tem um formato JSON e para cada uma das variáveis existentes no modelo irá ter:

```
"email":{
    "how_to_find":"id",
    "what_to_find":"email",
    "what_to_do":"sendKeys",
    "type_of_action":"textbox"
  }
```

Listagem 5: Formato do ficheiro de mapeamento de variáveis do modelo

- **how_to_find** - com que atributo queremos que seja encontrado o elemento HTML na aplicação. Pode tomar os seguintes valores: *id*, *xpath*, *cssSelector*, *className*, *linkText*, *name*, *tagName*, *partialLinkText*.
- **what_to_find** - o valor do atributo. Pode tomar qualquer tipo de valores.
- **what_to_do** - qual o tipo de ação que irá ser executada. Pode tomar os seguintes valores: *sendKeys*, *click*, *submit*, *moveToElement*.

- **type_of_action** - serve para saber qual o tipo de elemento HTML. Pode tomar os seguintes valores: *textBox*, *selectBox*, *checkBox*.

Na Listagem 5, para a variável **email** o elemento HTML deve ser encontrado através do **id**, que deve ser "**email**" e a ação a realizar para ele será um sendKeys (que é introduzir um valor de *input*) e o tipo do elemento é um textbox.

Os valores de *input* são guardados noutro ficheiro de configuração, com um formato JSON, apresentado na Listagem 6 onde é possível ver que a variável **email** será preenchida com o valor *"email_teste@gmail.com"*.

```
{
  "email":"email_teste@gmail.com"
},
{
  "email_op":"raphaeljr28@gmail.com"
},
```

Listagem 6: Ficheiro de valores de *input*

O ficheiro de configuração de *input* tem o seguinte formato (apresentado na Listagem 7):

- $model\_name$: o nome do elemento do modelo
- $value$: o valor de *input* que queremos ter no elemento HTML nos casos de teste gerados

```
{
    $model_name$: $value$
}
```

Listagem 7: Formato de valores de *input*

5.3.1 Demonstração

A ferramenta desenvolvida a partir dos dois ficheiros descritos acima, gera os casos de testes. Nas próximas secções serão explicados com mais detalhes cada um dos passos do processo de geração de testes.

Vejamos um exemplo concreto de configurações para o elemento HTML apresentado na Listagem 8.

```
<input id=”FromTag” class=”keyValue span span24 required” name=”origin” type=”text’>
```

Listagem 8: Elemento *HTML* correspondente ao elemento *from* do modelo

Na Listagem 9 é apresentada a configuração de mapeamento, em que é dito que o elemento HTML (apresentado na Listagem 8) vai ser encontrado através do seu *id* com o valor *FromTag* em que a acção a realizar é de *sendKeys*.

```
”from”:{
        ”how_to_find”:”id”,
        ”what_to_find”:”FromTag”,
        ”what_to_do”:”sendKeys”,
        ”type_of_action”:”textbox”
      }
```

Listagem 9: Exemplo de ficheiro de mapeamento

Na Listagem 10 é apresentado uma parte do ficheiro de *inputs*, em que é dito que o valor *from* tomará o valor de *”Porto”*.

```
{ ”from”:”Porto” }, ...
```

Listagem 10: Exemplo de ficheiro de valores de *input*

Na Figura 5.2 é possível ver na parte esquerda o ficheiro de mapeamento, na parte direita o ficheiro de *inputs* e na zona do meio o código gerado conforme ambos os ficheiros de configuração e onde facilmente se percebe a relação entre o código gerado e os ficheiros de configurações.

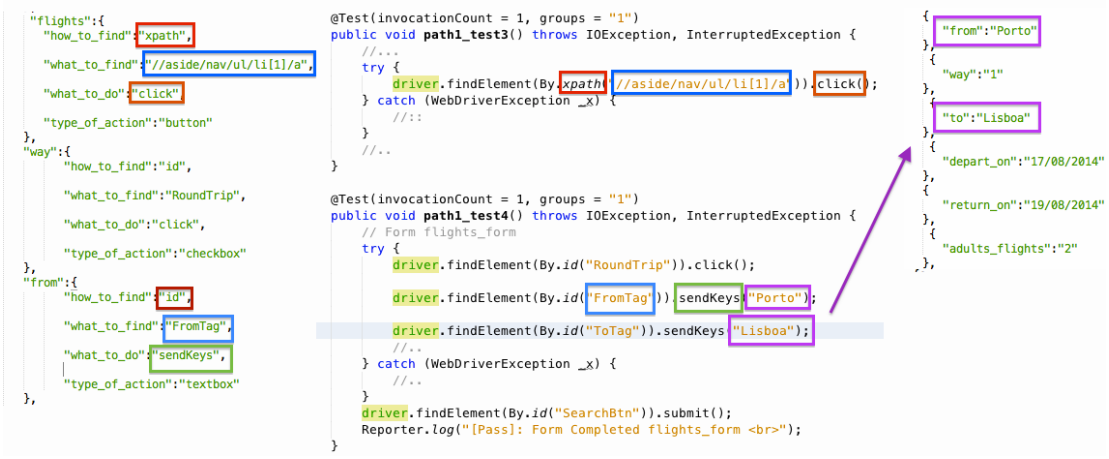

Figura 5.2: Código gerado a partir dos ficheiros de configuração

5.3.2 Catálogo de validações possíveis

Foram ainda implementadas vários tipos de validações relevantes nos testes sobre interfaces gráficas. Este tipo de validações pode ser bastante útil para verifica alguma propriedades da interface gráfica.

Nos testes de software, precisamos verificar se a saída de caso de teste contra um conjunto predefinido de dados de teste. Estas validações fornecem um forma ágil de conseguir essas verificações. As validações são introduzidas no modelo e na geração de casos de teste serão convertidas para instruções *Selenium*. No modelo as validações são representadas pelas *tags* '<onexit/>' e '<onentry/>'. Sempre que dentro de um estado existir uma *tag* '<onexit/>' significa que sempre que se sair desse estado irá ser feita uma validação.

- Verificar se um elemento está visível – **displayed?/not_displayed?**.
- Verificar se um elemento está *enabled* ou *disabled* – **enabled?/disabled?**.
- Verificar se elemento está selecionado(*selectbox/chekcbox*) – **is_selected**.
- Verificar a existência de certos elementos – **is_present?/not_present?**
- Verificar url de uma página – **url**.
- Verificar a propriedade de CSS(como por exemplo, *background-color*, *position*, etc) de um elemento – **css**
- Verificar um certo atributo de um elemento (por exemplo, o atributo value de um input) – **attribute**.
- Verificar se um certo elemento possui uma *string* – **default**.

- Verificar se um certo elemento contém uma certa *string* – **contains**.
- Verificar se um elemento contém uma *string* que corresponde a uma expressão regular – **regex**.

Na Figura 5.3 é possível ver que no modelo existe uma *tag* '<onexit id="message_authentication" type="default"/>'. Neste caso, será feita uma validação do tipo **default** no elemento *message_authentication*. Aquilo que será validado será o texto presente num elemento ('ui-pnotify-text') do SUT e em que a mensagem será que terá que aparecer será 'Autenticação efetuada com sucesso.'. No final da Figura 5.3, é possível ver o código gerado a partir de modelo, do ficheiro de mapeamento e do ficheiro de valores para esta validação.

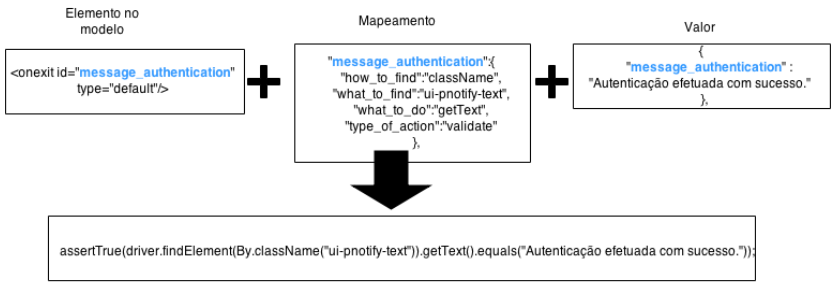

Figura 5.3: Exemplo de uma validação

Tendo estes requisitos preenchidos (modelo e os ficheiros de configuração) a ferramenta será capaz de gerar os casos de teste. Nas próximas secções serão introduzidas qual o processo pelo qual passam estes ficheiros até que se cheguem à geração dos testes.

5.4 Transformação do modelo para grafo

O primeiro passo para começar a geração de casos de teste é gerar a partir do modelo um grafo que contém toda a sua informação. Nesta secção irá ser explicado como o modelo previamente construído irá ser transformado num grafo.

Na teoria dos grafos, um multigrafo é um grafo que permite arestas com os mesmos vértices iniciais e finais, fazendo com que dois vértices possam estar ligados por mais do que uma aresta. É definido por

$$G := (V, A, s, t)$$

onde V é um conjunto de vértices, A é um conjunto de arestas, s é uma função

$$s : A \to V$$

que atribui a cada aresta o seu vértice de origem e t é uma função

$$t : A \to V$$

que atribui a cada aresta o seu vértice de destino.

Foi necessário um multigrafo já que estes permitem ter a possibilidade de criar ciclos, possuir arestas em que o vértice de origem é o mesmo que o de destino, que o que numa aplicação web pode existir (por exemplo em clicks ou eventos que voltem para a mesma ou pedido AJAX que não implicam uma mudança de página). Foi realizado um *parser* (ver no anexo 6 mais detalhes) para transformar o modelo num grafo de forma a ter uma estrutura mais fácil para aplicar algoritmos de travessia sobre eles. Para a guardar o grafo foi utilizada uma biblioteca de grafos chamada JGraphT, que fornece várias estruturas de dados para guardar diferentes tipos de grafos e possui ainda vários algoritmos de travessia de grafos. Na Figura 5.4 é ilustrada um exemplo de um grafo gerado a partir do modelo em SCXML.

O recurso à teoria dos grafos torna mais fácil encontrar caminhos. Neste grafo gerado, cada um dos vértices representa uma página e as arestas representam acções (por exemplo, um click, a submissão de um formulário). A partir do SCXML será gerado um multigrafo direccionado em que estão representados todas as possíveis acções na GUI.

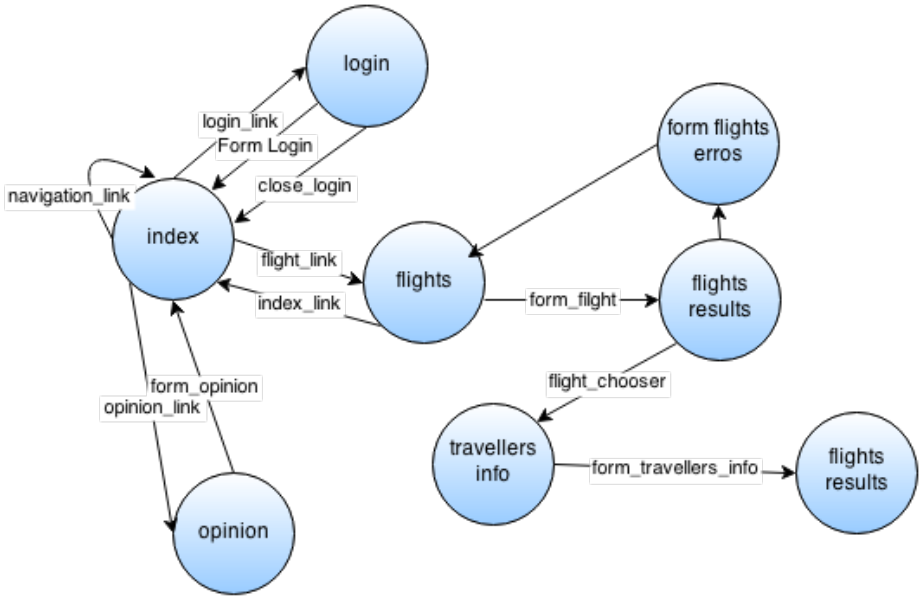

Figura 5.4: Grafo parcial gerado a partir do modelo

5.5 Geração de testes abstractos

De seguida, a geração de caminhos é feito através de um algoritmo de travessia sobre o grafo (ver capítulo 6). Podemos olhar para os testes sobre uma interface gráfica como um caminho, que é uma sequência de acções que são executadas sobre um conjunto de páginas web. Cada caminho terá múltiplos casos de teste. Na Figura 5.5 é representado cada um dos passos de um caminho. A partir de cada um desses passos e da sua informação irão ser gerados os casos de teste executáveis.

Este passo de geração de testes abstractos é independente da tecnologia em que irão os testes ser executados, o que permite uma maior flexibilidade e melhor manutenção. Por exemplo, se agora em vez de gerar os testes com instruções *Selenium* fosse necessário gera as instruções noutra ferramenta.

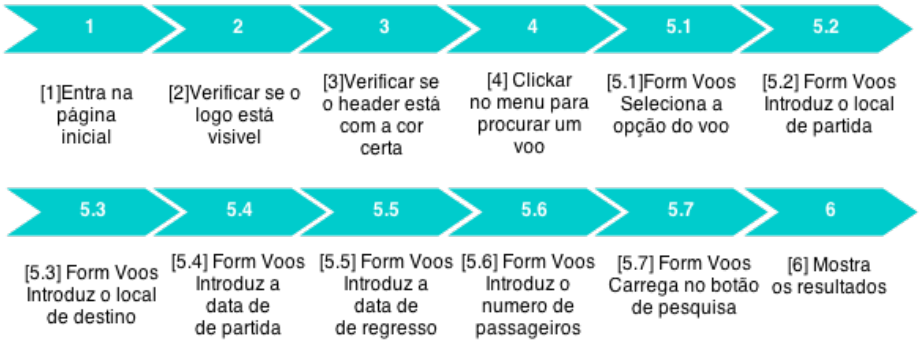

Figura 5.5: Caminho de um caso de teste

5.5.1 Critérios de cobertura

A geração de testes acaba por estar reduzida a um problema de travessia sob grafos. Sendo que o engenheiro de testes será livre de tomar a decisão sobre qual o algoritmo a usar para a geração de testes.

A diferença dos critérios entre os algoritmos de travessia sobre grafos, é que cada um deles possui uma forma diferente de selecionar os caminhos a serem testados e gerar os casos de teste que exercitem esses caminhos. É possível ter diferentes necessidades de testes, por exemplo, escolher um critério de cobertura que cubra todas as transições possíveis, que passe pelo menos uma vez por todos os vértices ou até necessidades mais específicas. A seguir estão descritos alguns desses critérios:

- critério que cubra todas as transições.
- critério que cubra todos os estados.
- critério que dado um estado inicial e um final, devolva caminhos entre eles (se existirem).

- critério que devolva caminhos aleatórios a partir de um estado.
- critério que não tenha repetições de estados.

5.6 Geração de casos de teste executáveis

A partir dos casos de testes abstractos, serão gerados os ficheiros dos casos de teste executáveis. O algoritmo de geração de casos de teste percorre cada um dos passos do caminho abstracto transformando-o e gerando o seu respectivo código. O algoritmo para realizar este processo possui um mapeamento para cada um dos elementos do caminho abstracto para código Java e instruções Selenium. Este algoritmo precisa do caminho abstracto e dos ficheiros de configuração para ir buscar os dados para as instruções Selenium geradas.

De modo a seguir as boas práticas relativas aos testes de software como, por exemplo, criar casos de testes pequenos e que indiquem claramente aquilo que fazem, que possam ser reutilizáveis, para cada caminho gerado anteriormente irá ser criado um grupo de testes, em vez de ter toda a sequência dos testes num só método. Em cada um desses grupos irão existir vários casos de teste separados em métodos. Deste modo os testes são mais fáceis de manter, de gerir e são mais facilmente alterados se houver a necessidade de o fazer. Na Figura 15 é possível ver o um trecho do código de um caso de teste gerado pela ferramenta para preencher o formulário de pesquisa de voos. É possível ver que a estrutura é semelhante a um método de Java normal e possui anotação de TestNG/JUnit.

```
@Test(invocationCount = 1, groups = "1")
public void path1_test4() throws IOException, InterruptedException {
    //..
    driver.findElement(By.id("RoundTrip")).click();
    driver.findElement(By.id("FromTag")).sendKeys("Porto");
    driver.findElement(By.id("ToTag")).sendKeys("Lisboa");
    driver.findElement(By.id("DepartDate")).sendKeys("17/08/2014");
    driver.findElement(By.id("ReturnDate")).sendKeys("19/08/2014");
    new Select(driver.findElement(By.id("Adults"))).selectByVisibleText("2");
    new Select(driver.findElement(By.id("Childrens"))).selectByVisibleText("1");
    new Select(driver.findElement(By.id("Infants"))).selectByVisibleText("0");
    //..
    driver.findElement(By.id("SearchBtn")).submit();
```

```
    //..
}
```

Listagem 11: Parte de código gerado

Na Figura 12 é representada uma das instruções de Selenium geradas. Aquilo que esta instrução faz é dizer a variável *driver* (que contém o *webdriver*) para encontrar o elemento *ReturnDate* através do *id* e enviar-lhe o valor *"19/08/2014"*. O que na prática significa que vai preencher um campo de texto com esse valor.

```
driver.findElement(By.id("ReturnDate")).sendKeys("19/08/2014");
```

Listagem 12: Instrução Selenium

Na Figura 5.6 é possível ver o código gerado (na parte direita) para preencher e submeter o formulário da interface gráfica (na parte esquerda).

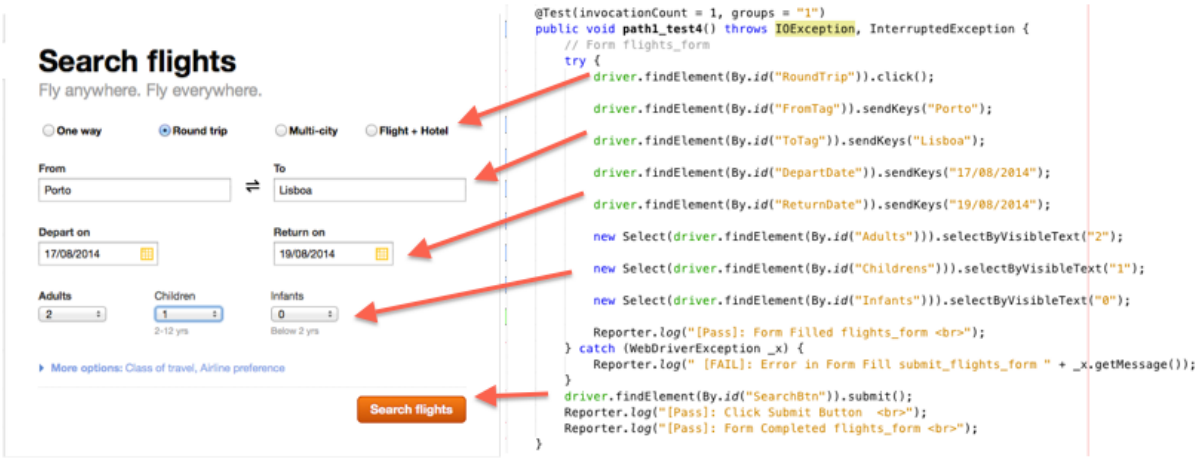

Figura 5.6: Execução do teste para o formulário de voos

Este modulo de geração de casos de teste gera não apenas instruções Selenium mas também gera métodos de teste organizados com anotações, gera código para os logs, gera código para excepções.

O *Logging* é o processo de registar acções. Fornece assim informação útil sobre aquilo que aquilo que foi executado. Permitem criar registros de todas as informações relevantes sobre a execução de um teste, onde se poderá verificar os resultados das suas execuções. Tornando mais fácil a investigação de problemas. Por exemplo quando um teste falha, pode ser útil ter um sistema de *logs* para poder rapidamente perceber onde esse teste falhou. As excepções são introduzidas devido ao facto de quando uma instrução *Selenium*

falha é lançada uma excepção em especifico. Portanto a melhor forma de saber qual a causa dessa falha é introduzir *trycatch* para as diferentes excepções possíveis de serem lançadas.

É possível ver um ficheiro de testes completo em anexo. Uma das vantagens para a geração do código desta forma é que os engenheiros de testes podem aceder a eles, perceber o que faz e executar apenas umas partes por exemplo.

5.6.1 Estrutura dos casos de teste

Os ficheiros de testes gerados possuem a estrutura descrita a seguir. A *framework* TestNG permite executar grupos de teste. Neste caso cada caminho terá um grupo de testes. No ficheiro de teste será gerado um método $init\_selenium()$ com uma uma anotação $@BeforeGroups$ para que o método seja invocado antes da execução de cada um dos grupos de teste. O objectivo deste método é iniciar o *WebDriver* em que serão executados os testes. Depois para cada passo do caminho será gerado um método que irá conter o caso de teste. Existe ainda um método $closeBrowsers$ (Figura 13) que irá fechar o *Webdriver*, este método possui uma anotação $@AfterGroups$ (Figura 14) para que seja executado depois de cada um dos grupos.

```
@BeforeGroups(groups = {"1","2","3","4"})
public void init_selenium() {
    FirefoxProfile profile = new FirefoxProfile();
    this.driver = new FirefoxDriver(new FirefoxBinary(new File("/Applications/Firefox.app/Contents/MacOS/firefox-bin")), profile);
    this.driver.get("https://twitter.com/");
    this.driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
    WebDriverWait wait = new WebDriverWait(driver, 60);
}
```

Listagem 13: Código executado antes de cada grupo de testes

```
@BeforeGroups(groups = {"1","2","3","4"})
public void closeBrowsers() {
    this.driver.close();
}
```

Listagem 14: Código executado depois de cada grupo de testes

5.6.2 Configuração da geração de casos de teste

De forma a ser possível escolher diferentes opções para a geração de casos de teste, foram disponibilizadas algumas opções para o engenheiro de teste escolher quando a ferramenta de geração de casos de testes é executada. Este componente implementado serve para configurar alguns dos seguintes critérios:

- seleccionar o *browser* onde irão correr os testes
- selecionar se de cada vez que uma página é acedida sejam verificados links que levam a páginas que não existem (e,g *broken links*)
- verificar se todas as imagens são mostradas correctamente e que possuem um *href* correcto
- selecionar se cada vez que há um erro na aplicação que está a ser testada se pretende tirar um *screenshot* à página.
- seleccionar qual o algoritmo de travessia a usar.
- seleccionar o tipo de mutações a realizar (que serão abordadas na secção 5.8)

Estas opções vêm enriquecer e completar um pouco os casos de teste gerados. É ainda possível escolher quais os ficheiros de teste que se quer gerar, que são os seguintes:

- Normal, onde será gerado os casos de teste normais (sem qualquer tipo de mutações)
- Com todas as mutações feitas de forma aleatória onde é gerado um ficheiro com cada uma das mutações.
- Um ficheiro que tenha contenha as mutações feitas a partir de um ficheiro de configuração.

5.7 Execução dos Testes e Avaliação

Uma vez gerados os casos de testes, estão prontos a serem executados. Como referido anteriormente as *frameworks* utilizadas para executar os casos de teste gerados são o *Selenium* e o *TestNG*.

A execução dos testes simula a interacção entre um utilizador e o *browser*. No final dos testes será gerado um ficheiro de *log*, que irá conter toda a informação detalhada sobre os casos de teste realizados, de forma a permitir analisar o resultado de cada um desses testes. Se alguma anomalia ocorrer é possível visualizar o que aconteceu durante a sua execução. Por exemplo, uma anomalia pode ser, um erro resultante de se tentar clicar num elemento HTML que não estaria visível.

Com a *framework* de TestNG é gerado um ficheiro HTML (Figura 5.7) com um conjunto de estatísticas (data de execução, tempos de cada um dos testes, ordem cronológica de execução, numero de testes que passaram, etc). Neste ficheiro existe uma zona onde é mostrado o *Reporter Output* que é para onde é enviado todo o *log* de cada um dos casos de teste. Como se pode ver pela Figura 5.7, cada caso de teste possui o seu *log* permitindo ter uma representação mais visual e organizada. Tornando, assim mais fácil a analise de cada um dos casos de teste.

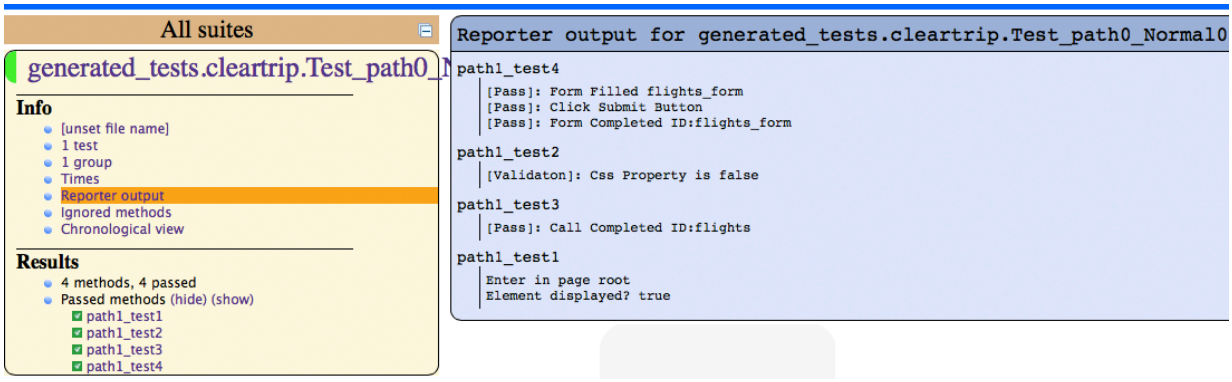

Figura 5.7: Log do cleartrip

Este relatório é também uma forma de documentar a execução de teste e pode ser apresentado de uma forma personalizável sendo apenas necessário alterar o ficheiro de CSS, desta forma podem uniformizar a visualização destes resultados com outro tipo de *log's* que possam estar mais familiarizados.

Na analise dos resultados, o engenheiro de testes poderá de realizar uma análise de cada um dos casos de teste executados (Figura 5.7).

5.8 Mutações

Como falado no capítulo anterior, as mutações servem para introduzir erros para perceber qual o comportamento da aplicação. Foram implementadas todas as mutações que foram anteriormente referidas. Desta forma, é possível gerar casos de teste que contenha cada uma mutações, feitas de forma aleatória. Foram implementadas as seguintes mutações:

- *Lapse*
- *Slip*
- *Mistake*
- *Double_click_sumbit* - realiza um duplo clique num botão de submissão dos formulários.
- *Remove_required_field* - remove um campo obrigatório de um formulário.
- *Double_click_call* - realiza um duplo clique em vez de um clique simples.
- *Double_click_menu_call* - realiza um duplo clique num link.

- *Inject_back_event* - injecta um evento de voltar atrás numa página.
- *Inject_refresh_event* - injecta um evento de *refresh* numa página.

No entanto, por vezes o engenheiro de testes pode ter a necessidade de escolher quais e onde quer, que as mutações sejam efectuadas. Isso pode ser feito através da criação de um ficheiro de configuração de mutações. O ficheiro terá o formato apresentado na Listagem 15, em que:

```
{
   "type" : "lapse",
   "model_element": "email",
   "fail" : "1"
}
```

Listagem 15: Exemplo do ficheiro de mutações

- **type** define qual o tipo de mutação que se quer realizar. O conjunto de valores possivéis para este atributo são as seguintes:
  - *lapse*
  - *slip*
  - *mistake*
  - *double_click*
- **model_element** define qual elemento do modelo se quer realizar a mutação.
- **value** é um campo que não é obrigatório e apenas é necessário a mutação é feita quando a mutação é feita num valor de *input* do utilizador.
- **fail** permite para saber se a mutação irá fazer o teste falhar ou seja, gerar um teste inválido. Com o valor **1** significa que a mutação irá fazer o causar uma falhar, com o valor **0** significa que a mutação não irá fazer com que o teste falhe.

Vamos agora exemplificar a realização de mutações a partir de um ficheiro de configuração. Na Listagem 16 é apresentado um dos elementos de um ficheiro de mutações.

```
{    "type" : "mistake",
     "model_element": "from",
     "value":"oprto",
     "fail" : "1"
}
```

Listagem 16: Exemplo do ficheiro de mutações

Esta mutação vai fazer com que o elemento **from** do modelo sofra uma mutação do tipo **mistake** (que modificam valores de *input*) com o valor "oprto". Esta mutação irá resultar o código gerado na Listagem 17.

```
@Test(invocationCount = 1, groups = "1")
   public void path1_test4() throws IOException, InterruptedException {
      try {
            Reporter.log(" Mutation Mistake From File in from <br>");

            driver.findElement(By.id("RoundTrip")).click();

            driver.findElement(By.id("FromTag")).sendKeys("oprto");
            String from = driver.findElement(By.id("FromTag")).getAttribute("value");

            driver.findElement(By.id("ToTag")).sendKeys("Lisboa");
            String to = driver.findElement(By.id("ToTag")).getAttribute("value");
            ...
            ...
            ...

 }
```

Listagem 17: Código gerado com a mutação feita a partir do ficheiro de mutações

Como é possível ver pelo modelo (ver Figura 5.8) o elemento **from** pertence a um formulário e é obrigatório. Ao introduzirmos esta mutação faremos com que a submissão do formulário falhe e que vá para um estado de erro **flights_form_error**. Este estado **flights_form_error** possui duas validações que serão

realizadas sempre que se entrar nele. Uma é verificar a cor da caixa de erro e a outra é verificar se a mensagem de erro é a correcta.

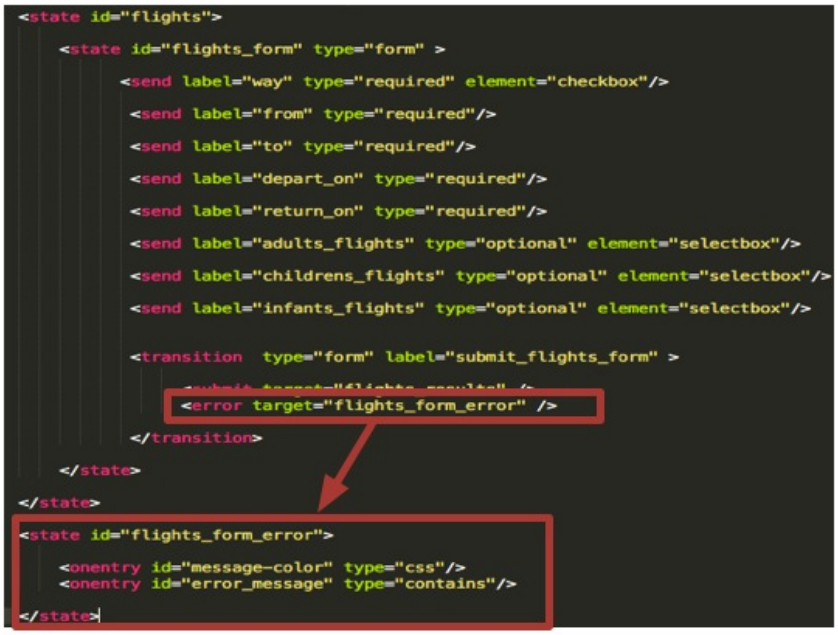

Figura 5.8: Representação do formulário de pesquisa de voos e respectivo estado de erro

Portanto o resultado esperado desta mutação é fazer com que o formulário falhe e que verifique se as mensagens de erro mostradas estão correctas. Pela Figura 5.9 podemos verificar qual o estado da aplicação depois de ter sido feita a mutação e submetido o formulário. No final o caso de teste passou já que ambas as verificações no estado de erro foram validadas, ou seja, a mensagem de erro estava correcta e o fundo tinha a cor certa.

5.9 Conclusão

Neste capítulo foi apresentada a abordagem proposta para testes baseados em modelos para interfaces gráficas. Foi descrito todo o processo e a arquitectura da solução implementada. A ferramenta foi descrita de modo a detalhar seu funcionamento.

Como explicado anteriormente, no processo de MBT há necessidade de uma linguagem abstracta para representar o modelo, na abordagem proposta é usado o SCXML. Há também a necessidade de geração de casos de teste, que na nossa abordagem passa por transformar a máquina de estados num grafo, de seguida utilizar a teoria de grafos para encontrar caminhos possíveis e por fim transformar esses caminhos

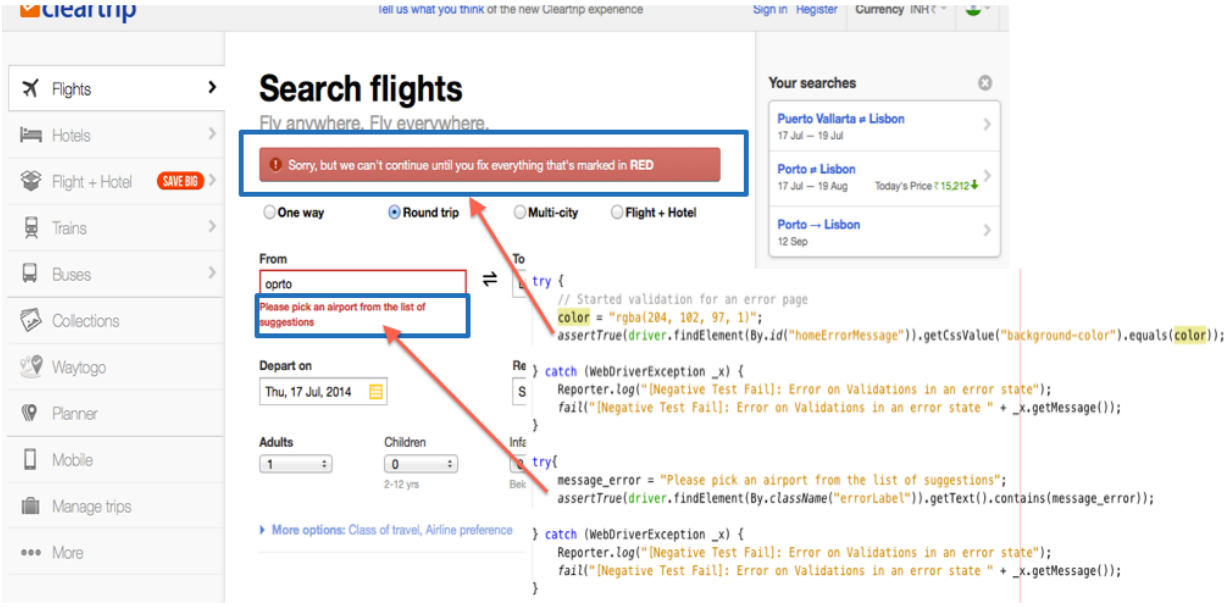

Figura 5.9: Estado do formulário depois de ter sido feita a mutação

em casos de teste para que possam ser executados.

O modelo foi usado como oráculo e os ficheiros de configuração para completar os casos de teste gerados. Mais exemplos serão dados no capítulo ??.

6 | Implementação da abordagem para testes baseados em modelos

6.1 Arquitectura da solução

A arquitectura proposta foi pensada para ser modelar e cada um dos módulos funcionar de forma independente. Devido ao facto de ser um trabalho de investigação feito de raiz e criado para ser integrado com outras soluções (como por exemplo, a ferramenta PARADIGM), será mais passível de sofrer alterações ao longo do projecto. É por isso, essencial tentar minimizar ao máximo cada uma das mudanças que possam ser feitas na arquitectura ou até dos requisitos. Foi por isso dada uma atenção especial nesse aspecto, de tentar criar um software que seja fácil de manter e que possa ser extensível e flexível. Para isso, foram criados módulos que funcionam de forma independente e usados vários padrões de desenho de forma a reduzir as dependências e coesão entre componentes tornando a manutenção e os seus testes mais fáceis. Os padrões de desenho são uma coleção de soluções para problemas conhecidos e recorrentes no desenvolvimento de software. Um padrão de desenho estabelece um nome e define o problema, a solução e quando aplicar esta solução e suas quais as suas conseqüências [30].

Desta forma a ferramenta implementada não fica apenas limitada ao tipos de modelos ou de ficheiros que estão a ser usadas nesta abordagem. Assim seria possível substituir o modelo em SCXML por outro formato qualquer, sem ser necessário alterar a arquitectura. Ou por exemplo, a partir de outra solução de MBT gerar um grafo parecido ao da solução implementada de forma a poder gerar casos de teste.

Nas próximas secções serão apresentados os packages e respectivas classes implementadas e quais as suas principais funcionalidades e responsabilidades. A figura 6.1 é representado a organização de classes e packages da solução.

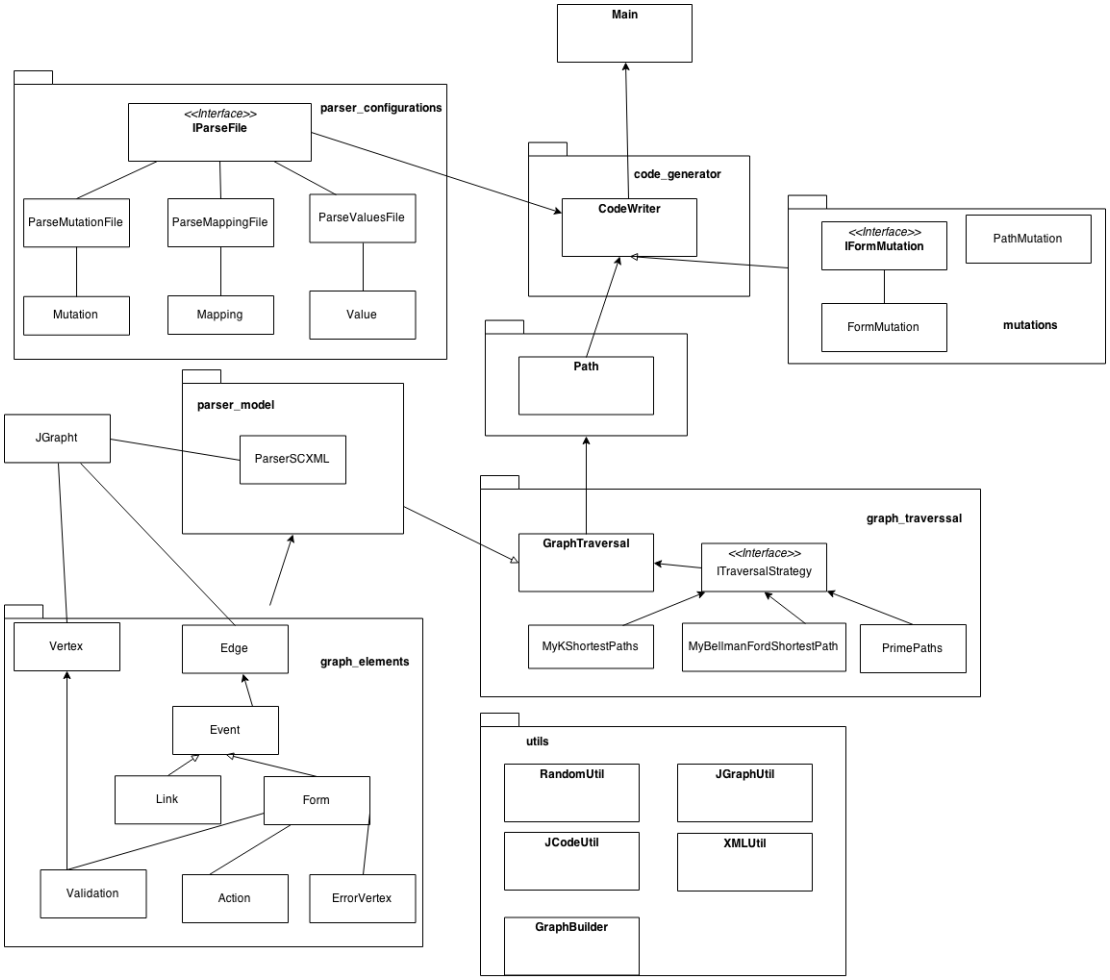

Figura 6.1: Organização de classes e packages

6.2 Packages

6.2.1 Package graph_elements

Este packages possui as classes que representam os objectos necessários para a criação do grafo a partir do modelo.

- Vertex - Representa uma página/estado e possui uma lista de validações quando se entre nele.
- Edge - Representa uma ligação entre dois vertices, cada objecto Edge possui um Event.
- Event - Classe abstracta que pode ser um Call ou um Form.

- Call - Representa uma conjunto de ações realizadas por um utilizador. É composto por uma lista de interacções(por exemplo : click num link, navegar por um menu e depois carregar num botão)
- Form - Representa um formulário numa página. Possui uma lista de acções(por exemplo preencher caixas de texto), possui um botão de submissão, possui um conjunto de validações e possui condições.
- Validations- Classe que possui uma lista de Validation e possui todos os métodos para adicionar as validações.
- Validation - Representam as validações disponíveis no modelo. Por exemplo, is_displayed?,not_displayed?,exists?,etc

6.2.2 Package parser_model

Responsável por fazer o parsing do modelo (SCXML) e construir um grafo. Para realizar o *parsing* do modelo é usado um padrão de estratégia. Desta forma, é possível alterar o tipo de modelo, sem necessitar de modificar quem irá usá-lo. Por exemplo, se houver uma mudança de requisitos e houver introdução de um modelo que seja representado noutro formato que não SCXML, apenas seria necessário criar um parser que retornasse um grafo e mudar a estratégia de Parsing. Permitindo assim, que seja mais fácil a aplicação mudar os tipos modelos utilizados.

6.2.3 Package parser_configurations

Possui as classes que irão fazer o parsing dos ficheiros de configuração necessários. É usado um padrão de estratégia tal como usado no *package parser_model*, para realizar o parsing dos ficheiros de configuração.

6.2.4 Package graph_traverssal

Depois de ter o grafo construído, existem vários algoritmos que se podem implementar para percorrer o grafo e gerar caminhos. Foi usado um padrão de estratégia para a escolha do algoritmo de travessia sobre o grafo. O padrão de estratégia é um padrão comportamental e é usado quando existem vários algoritmos possíveis de implementar para uma tarefa em especifica, sendo que o cliente pode escolher qual quer usar. Este padrão fornece uma maneira de definir um conjunto de algoritmos, encapsulados num objecto , tornando assim os algoritmos permutáveis. Assim é possível implementar vários outros algoritmos sem haver necessidade de alterar a implementação de quem os usa.

Foi usado um padrão de estratégia para a escolha do algoritmo de travessia sobre o grafo. O padrão de estratégia é um padrão comportamental e é usado quando existem vários algoritmos possíveis de implementar para uma tarefa em especifico, sendo que o cliente pode escolher qual quer usar. Este padrão

fornece uma maneira de definir um conjunto de algoritmos, encapsulados num objecto , tornando assim os algoritmos permutáveis. Assim é possível implementar vários outros algoritmos sem haver necessidade de alterar a implementação de quem os usa.

Foi implementado um algoritmo que a partir de uma nodo inicial, faz uma travessia por todos os vértices que saem deste nodo. Como referido anteriormente este tipo de grafo pode conter ciclos. Assim, é possível criar um limite do numero de vezes que um caminho pode passar por um Nodo e um limite do numero de vezes que um caminho pode passar por um certo vértices.

A menos que seja limitado pelas restrições nos números de visitas a cada vértice ou aresta, o algoritmo garante que todos os vértices e todas as arestas são percorridas pelo menos uma vez. Garantido, assim, uma cobertura total da aplicação por parte dos testes gerados.

Este algoritmo foi implementado para cobrir as principais necessidades actuais, já que satisfaz um critério rigoroso de todos os caminhos, com um limite definido pelo utilizador no número de iterações em cada vértice e aresta.

No anexo 6 é possível ver mais alguns algoritmos implementados.

6.2.5 Package code_generator

CodeWriter

Existe a classe *CodeWriter* que recebe a uma lista de caminhos, e os valores dos ficheiros de configuração e que constrói e gera o ficheiro com os casos de testes, para os respectivos caminhos e valores de configuração.

De seguida a classe *CodeGenerator* possui os métodos necessários para gerar o código para cada um dos objectos do caminho. Por exemplo qual o código a gerar para um formulário ou um clique num *link*.

Na classe *CodeWriterStatic* é gerado todo o código mais estático, e que não depende do caminho. Por exemplo nesta classe são gerados os métodos auxiliares para iniciar o *WebDriver*, gerar os imports, etc.

Esta separação foi feita de forma a separar aquilo que estático a cada uma das gerações de casos de teste e aquilo que depende do conteúdo de cada um dos caminhos.

Para a geração do código foi utilizada uma libraria chamada *JCodeModel*, que permite gerar código fonte em Java. De forma a complementar e organizar os casos de testes gerados em Selenium, foi introduzida a *framework* TestNG/JUnit, que permite organizar os testes e criar relatórios de forma mais fácil. Ambas as *frameworks* completam-se (Selenium liga-se com o *browser* e o TestNG/JUnit organiza os testes).

6.2.6 Package mutations

Neste packages encontram-se as classes responsáveis por gerar mutações nos casos de testes. Por exemplo possui a classe FormMutation, que efectua vários tipos de mutações para os formulários.

6.2.7 Package util

Este package contem várias classes úteis e reutilizáveis.

7 | Exemplos de aplicação

Neste capítulo serão apresentados exemplos de aplicação para demonstrar o funcionamento da ferramenta desenvolvida. Para cada um dos exemplos foi desenvolvido um modelo, um ficheiro de configuração de mapeamento, um ficheiro com valores, e um ficheiro de mutações.

O primeiro exemplo é relativo à criação de um anúncio numa plataforma de para compra/venda de artigos para a area da agricultura, pesca e pecuária, o AgroSocial. Esta plataforma foi escolhida já que se trata de uma plataforma criada no âmbito de um projecto de mestrado em Engenharia Informática da Universidade do Minho, e estando na sua versão beta era interessante testá-la para conseguir encontrar eventuais erros. Uma das principais funcionalidade é permitir criar anúncios de compra e venda, por isso essa será a funcionalidade de exemplo. O segundo exemplo de aplicação realizado é sobre uma aplicação com milhões de acessos diários, o Twitter. O primeiro exemplo é sobre a capacidade de encontrar erros e o segundo mais sobre a capacidade de gerar muitos testes.

7.1 AgroSocial

O AgroSocial é uma rede social para promover a agricultura e produtos tradicionais, que visa agregar informação ligada à agricultura. É também uma plataforma onde é possível criar anúncios de compra e venda de produtos. A funcionalidade testada é criação de um novo anúncio na plataforma do AgroSocial.

7.1.1 Modelo

Na Figura 7.1 é ilustrado a máquina de estados para a interface gráfica do AgroSocial.

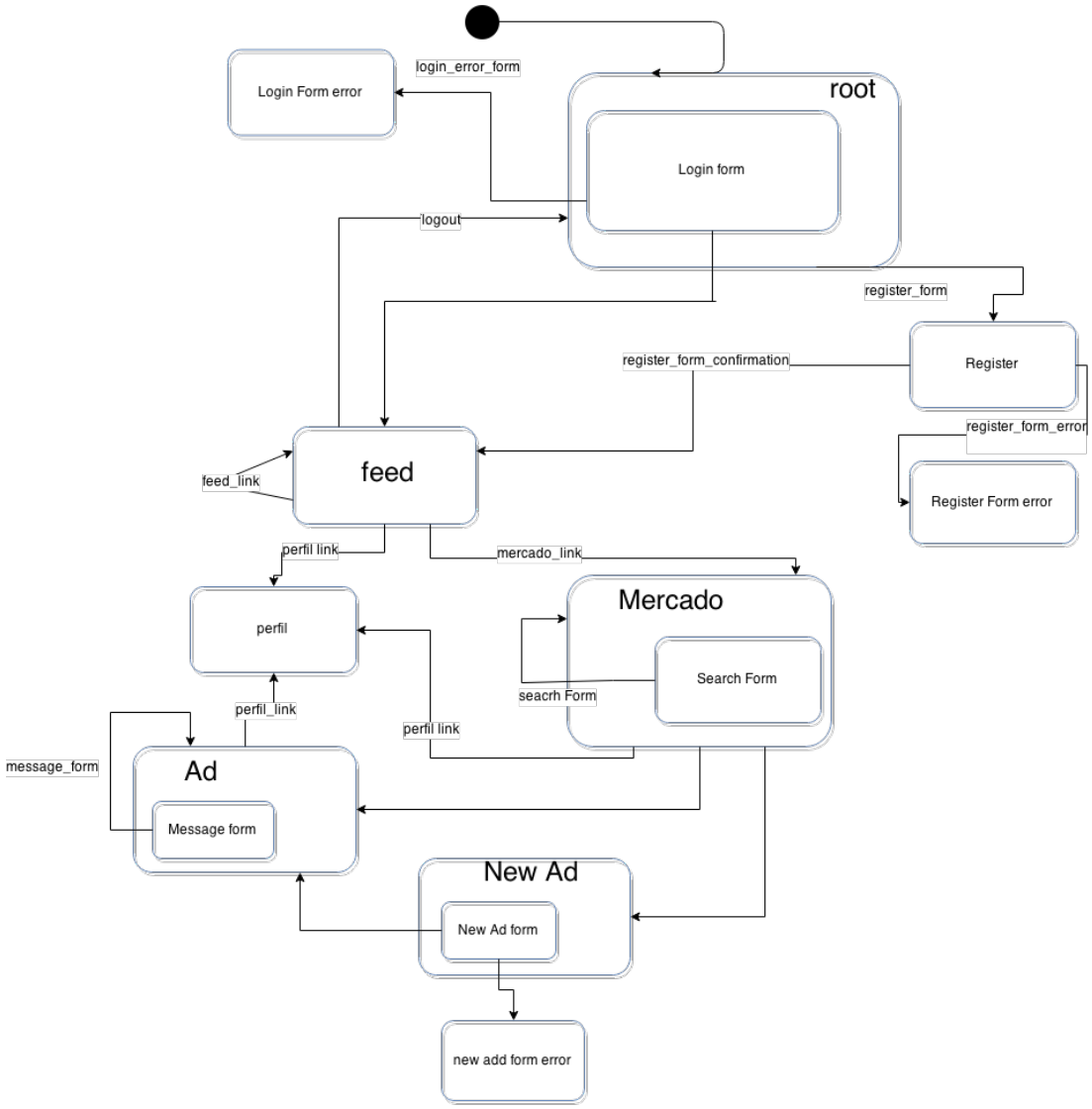

Figura 7.1: Máquina de estados AgroSocial

Na Listagem 18 é apresentado qual a representação do estado de crição de um novo anúncio ('new_ad') em SCXML. O estado *new_ad* representa uma página da aplicação e nessa página encontra-se um formulário, que é representado pelo subestado *new_ad_form*.

```
<state id="new_ad'>

  <state id="new_ad_form" type="form'>
    <send label="category" type="required" element="selectbox"/>
    <send label="title" type="required"/>
    <send label="description" type="required"/>
    <send label="price" type="required" />
```

```
    <send label="measure" type="required" element="selectbox"/>
    <send label="county" type="required" element="selectbox"/>
    <send label="local" type="required"/>
    <transition  type="form"  label="submit_new_ad_form" >
      <submit target="ads" />
      <error target="new_ad_error_form" />
    </transition>

    <!-- Verificar que anuncio tem o nome certo -->
    <onexit id="notification_new_ad" type="displayed?"/>
    <!-- Verificar notificacao -->
    <onexit id="ads_title" type="displayed?"/>
  </state>

</state>
```

Listagem 18: SCXML para respresentar o estado de criação de um novo anúncio

Na parte esquerda da Figura 7.5 é apresentado o formulário relativo à inserção de um anúncio. Os formulários em HTML são utilizados para enviar informação dos utilizadores para o servidor. Do lado direito é apresentado a parte do modelo que corresponde a este formulário. É representado no modelo por um estado (estado "new_ad_form") e dentro dele encontram-se cada um dos campos que são necessários preencher para completá-lo. O modelo completo para o AgroSocial pode ser encontrado no anexo A.1.

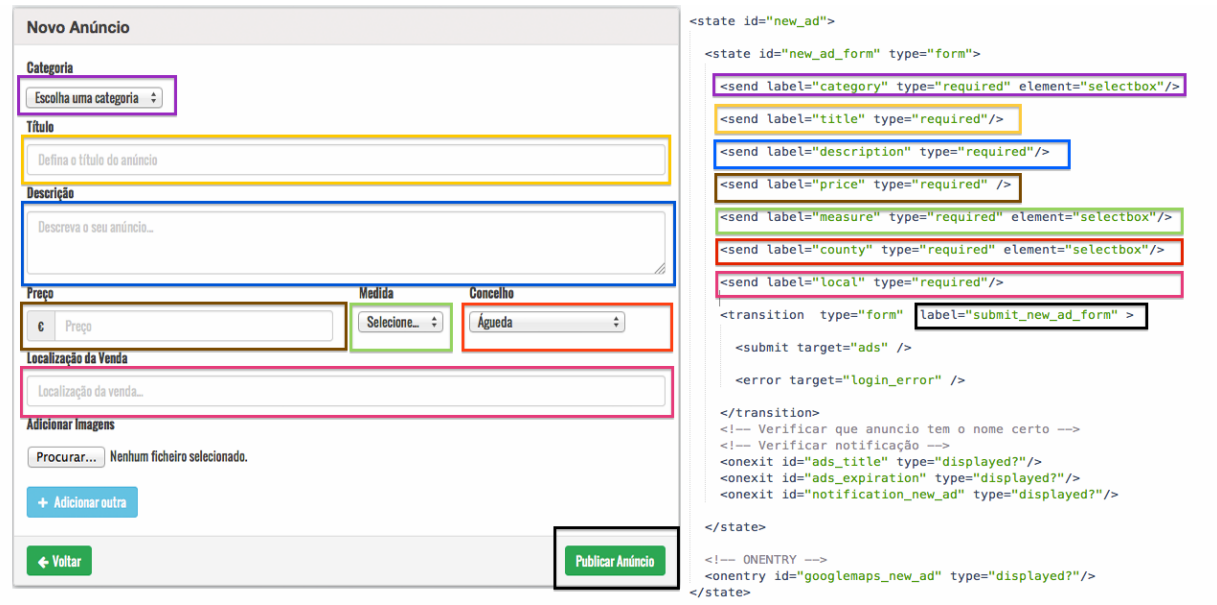

Figura 7.2: Interface gráfica para criação de novo anúncio e modelo em SCXML para a mesma

7.1.2 Ficheiros de Configuração

Os ficheiros de configuração contêm o mapeamento entre a interface gráfica e o modelo. A Figura 7.3 ilustra o ficheiro de mapeamento. Na parte esquerda o mapeamento e na parte direita os valores de *input*. O ficheiro de mapeamento irá ser necessário para completar as instruções de Selenuim para saber como encontrar os elementos e qual a acção a executar. TODO1- escrver um pouco mais sobre os ficheiros de config

Figura 7.3: Parte dos ficheiros de configuração (mapeamento e dados) para o AgroSocial

Na Listagem 19 é apresentado o parte do código gerado, onde é possível ver os valores que estão nos ficheiros de configuração mostrados anteriormente.

```
String category;
new Select(driver.findElement(By.id("ad_category_id"))).selectByVisibleText("Agricultura");
String title;
driver.findElement(By.id("ad_title")).sendKeys("Fruits and Vegetals");
title = driver.findElement(By.id("ad_title")).getAttribute("value");
```

Listagem 19: Código gerado para criar novo anúncio

7.1.3 Configuração

Como referido no capítulo 5. Antes de gerar os caminho de teste é possível escolher quais as configurações. Para este exemplo foi usada a seguinte configuração:

- Em que *browser* serão executados os testes - *Google Chrome*
- Tirar um screenshot à aplicação no momento da falha ou erro - Sim
- Escolher se sempre que se entra num novo estado, verificar se todos os links estão válidos - Sim
- Escolher se sempre que se entra num novo estado, verificar se todas as *sources* das imagens estão válidos - Sim
- Algoritmo de pesquisa de caminho a usar - usado o algoritmo que percorre todos caminhos, com um limite definido pelo utilizador no número de iterações em cada vértice e aresta.
  - Número máximo de visitas por vértice : 1
  - Número máximo de visitas por aresta: 2
- Escolher o tipo de mutações a gerar - todas as mutações foram seleccionadas. Ou seja, será criado um ficheiro diferente com o código gerado para cada uma das mutações.

7.1.4 Geração de casos de teste abstractos

Uma vez completado o modelo e os ficheiros de configuração, o processo de geração de testes pode começar. A ferramenta gerou um grafo a partir do modelo. A partir disso a ferramenta percorreu o grafo com a estratégia pretendida e retornou os caminho encontrados.

Foram encontrados 16 caminhos no total. Parte de um dos caminhos encontrados para a criação de um anúncio é ilustrado na Figura 7.4.

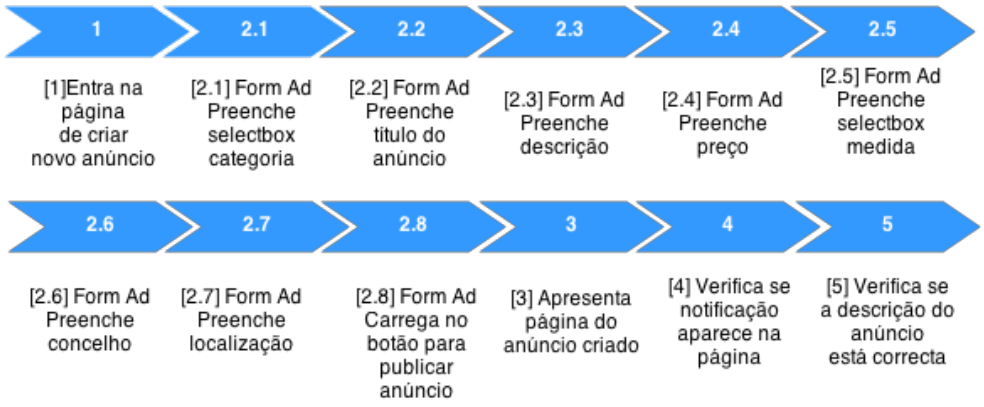

Figura 7.4: Caminhos encontrados

7.1.5 Geração de casos de teste executáveis

Uma vez criados os testes abstractos serão gerados os casos de teste executáveis a partir deles. O código apresentado na Listagem 20 é relativo ao caminho representado na Figura 7.4 do passo 2.1 até ao 2.8.

```
@Test(invocationCount = 1, groups = "5")
public void test6() throws IOException, InterruptedException{
    int gonna_fail = 0;
    // Form new_ad_form
    try {
            String category;
            new Select(driver.findElement(By.id("ad_category_id"))).selectByVisibleText("Agricultura");
            String title;
            driver.findElement(By.id("ad_title")).sendKeys("Fruits and Vegetals");
            title = driver.findElement(By.id("ad_title")).getAttribute("value");
            String description;
            driver.findElement(By.id("ad_description")).sendKeys("Fresh fruit and vegetals");
            description = driver.findElement(By.id("ad_description")).getAttribute("value");
            String price;
            driver.findElement(By.id("ad_price")).sendKeys("2");
            price = driver.findElement(By.id("ad_price")).getAttribute("value");
            String measure;
            new Select(driver.findElement(By.id("ad_type_price_id"))).selectByVisibleText("€/Kg");
            String county;
            new Select(driver.findElement(By.id("ad_city_id"))).selectByVisibleText("Braga");
            String local;
            driver.findElement(By.id("addresspicker_map")).sendKeys("Gualtar");
            local = driver.findElement(By.id("addresspicker_map")).getAttribute("value");
            Reporter.log("[Pass]: Form Filled new_ad_form <br>");
            gonna_fail = 0;
    } catch (WebDriverException _x) {
        Reporter.log(" [FAIL]: Error in Form Fill submit_new_ad_form " + _x.getMessage()  );
        fail(" [FAIL]: Error in Form Fill submit_new_ad_form " + _x.getMessage() );
```

```
    }
    driver.findElement(By.cssSelector("div.actions > input[name='commit']")).submit();
    Reporter.log("[Pass]: Click Submit Button  <br>");
    Reporter.log("[Pass]: Form Completed  ID:new_ad_form <br>");
    my_wait();
}
```

Listagem 20: Código gerado para criar novo anúncio

7.1.6 Execução e Avaliação

Uma vez os casos de teste executáveis gerados é possível executá-los. Na Figura 7.5 é possível ver parte do código gerado necessário para preencher o formulário para criar um novo anúncio e qual o estado do formulário da aplicação depois do código ter sido executado.

Na Figura 7.5 é mostrado o formulário depois de executado o código gerado para o preenchimento do formulário (Listagem 20).

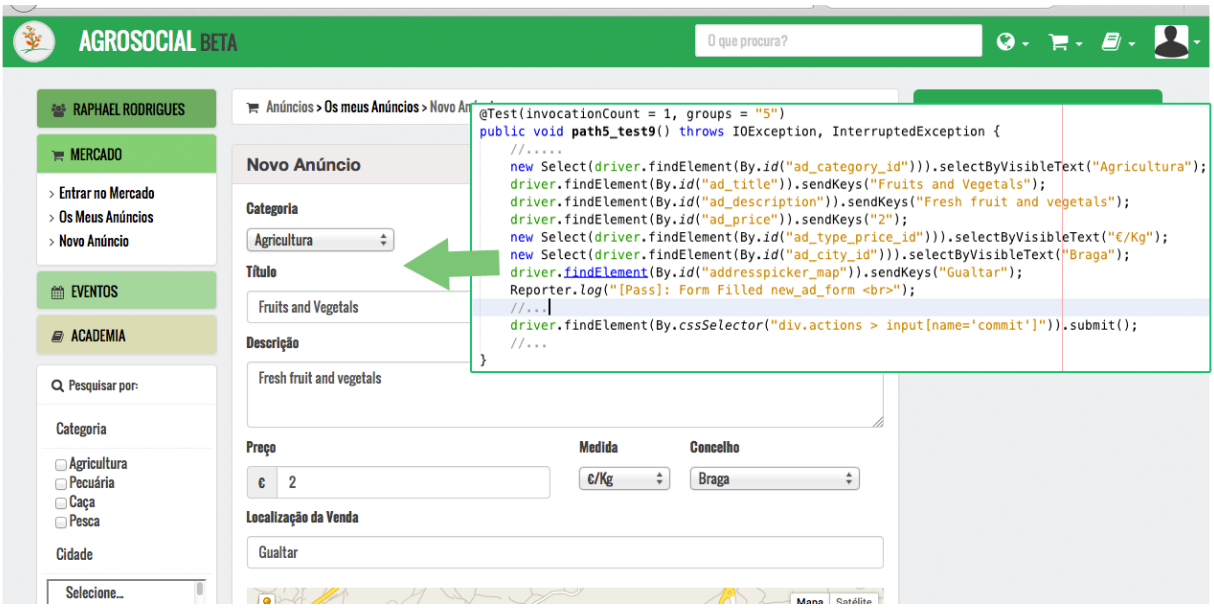

Figura 7.5: Interface gráfica para criar um novo anúncio e respectivo código gerado

Depois do caso de teste criar o novo anúncio, foram executados os restantes testes. No modelo foram introduzidas duas validações (explicadas na secção 5.3.2) depois de um anúncio ser criado. A Figura ?? mostra o estado de novo anúncio e é possível ver que possui os elementos de validação *<onexit>*, que irão gerar o código para as validações (ver Figura 7.6).

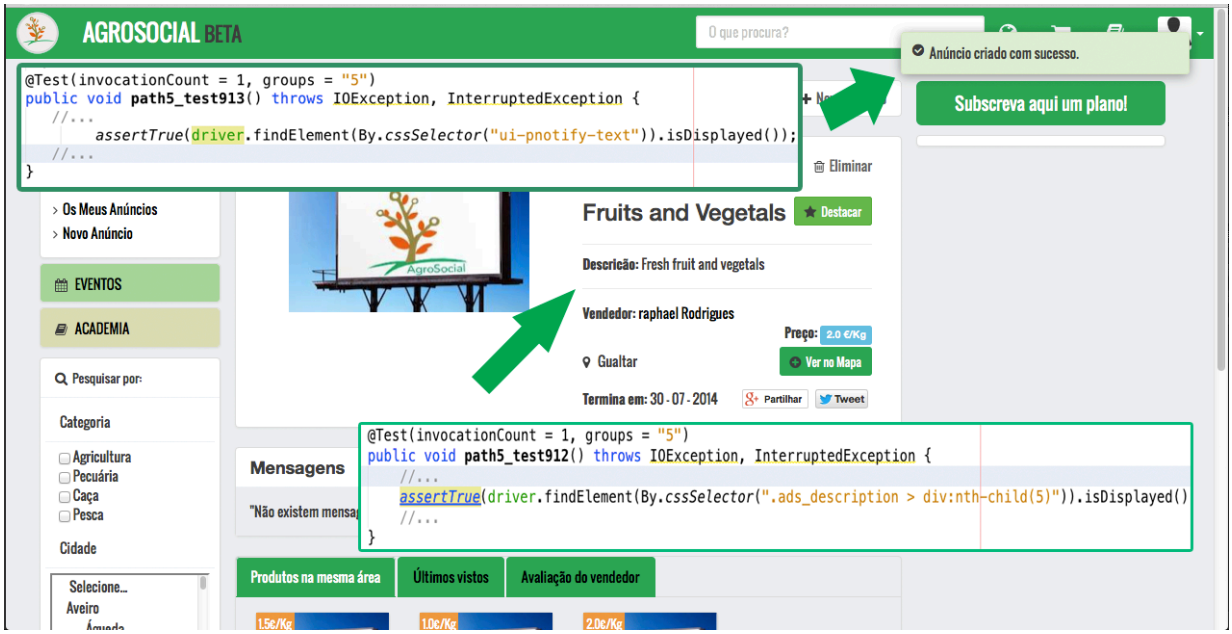

Figura 7.6: Resultado da criação de um novo anúncio e validações

Numa fase final, para o resultado da execução dos casos de teste é gerado um ficheiro *HTML* com os resultados, *log* e estatísticas. Terá de ser visto pelo engenheiro de testes para perceber se houveram falhas e onde é que elas aconteceram. Na Listagem 21 é possível ver o *log* de cada um dos testes executados e ver quais passaram e quais falharam. A Figura 7.7 mostra o ficheiro HTML com os resultados e os *logs*.

```
test1
Enter in page root
[Validaton]: Css Property is true

test2
[Pass]: Form Filled login_form
[Pass]: Click Submit Button
[Pass]: Form Completed ID:login_form

test3
[Validaton] Element displayed? true

test4
[Pass]: Click in a Menu message_hover
[Pass]: Call Completed ID:mercado

test5
[Pass]: Call Completed ID:new_ad

test6
[Pass]: Form Filled new_ad_form
```

```
[Pass]: Click Submit Button
[Pass]: Form Completed ID:new_ad_form

test7
[Validaton]: Element displayed? true

test8
[Validaton]: Element displayed? true
```

Listagem 21: Log para a execução de um novo anúncio

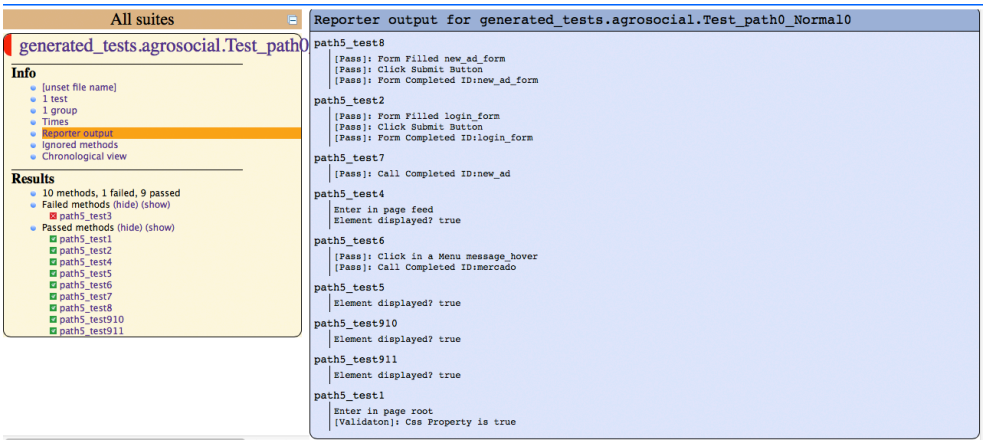

Figura 7.7: Log do teste

7.1.7 Mutações

A ferramenta desenvolvida disponibiliza várias opções para realizar mutações. Umas delas é realizar mutações a partir de um ficheiro. Neste caso o engenheiro de testes poderá criar um ficheiro onde pode escolher quais as mutações que quer realizar. Neste ficheiro é possível escolher se as mutações irão causar a falha do teste. No entanto nas mutações feitas de forma aleatória terá que ser feita uma análise manual dos resultados para verificar se cada mutação devia ter feito o teste falhar, ou não, e qual o resultado obtido.

Como exemplo, vai agora ilustrar-se a realização de uma mutação do tipo *lapse* (é realizada a eliminação de um das acções) num campo obrigatório, que fará com que a submissão do formulário falhe, logo terá de falhar e entrar no estado de erro do formulário. Na Figura 7.8 é possível ver o código gerado com uma mutação.

No Listagem 22 é apresentado o ficheiro onde as mutações a realizar. Neste caso podemos ver que o **model_element** *title* irá sofrer uma mutação do tipo *lapse* e que o teste irá de falhar. Este ultimo atributo de

saber se a mutação irá fazer com que o teste passe acaba por ser importante. Ao ser gerado o código para este elemento em especifico irá ter atenção neste atributo e gerar um teste negativo. Ou seja, a expectativa é que o teste com sucesso

Neste caso em especifico, o formulário irá ser preenchido sem o valor obrigatório *title*, fazendo com que ele não seja submetido com sucesso. Ao ser submetido irá ter de entrar num estado de erro do formulário. No entanto o código gerado irá depender deste valor. Então se o teste falhar é porque o formulário foi submetido com sucesso mesmo sem o preenchimento de um campo obrigatório. Por outro lado, se o teste passar com sucesso é porque o formulário não foi submetido com sucesso.

```
{
"type" : "lapse",
"model_element": "email",
"value":"mutation@gmail.com",
"fail" : "1"
},
{
"type" : "lapse",
"model_element": "title",
"fail" : "1"
},
{
"type" : "mistake",
"model_element": "last_name",
"fail" : "0"
},
```

Listagem 22: Ficheiro de configuração com mutações a realizar para o AgroSocial

Na Figura 7.9 é possível ver que o formulário não foi submetido com sucesso, já que a mutação eliminou a acção de preencher o campo título, que era obrigatório.

7.1.8 Erro detectado

Com a execução dos testes foi encontrado o seguinte erro na interface:

- Ao realizar um login ou ao registar deveria aparecer um notificação com uma mensagem dizendo que

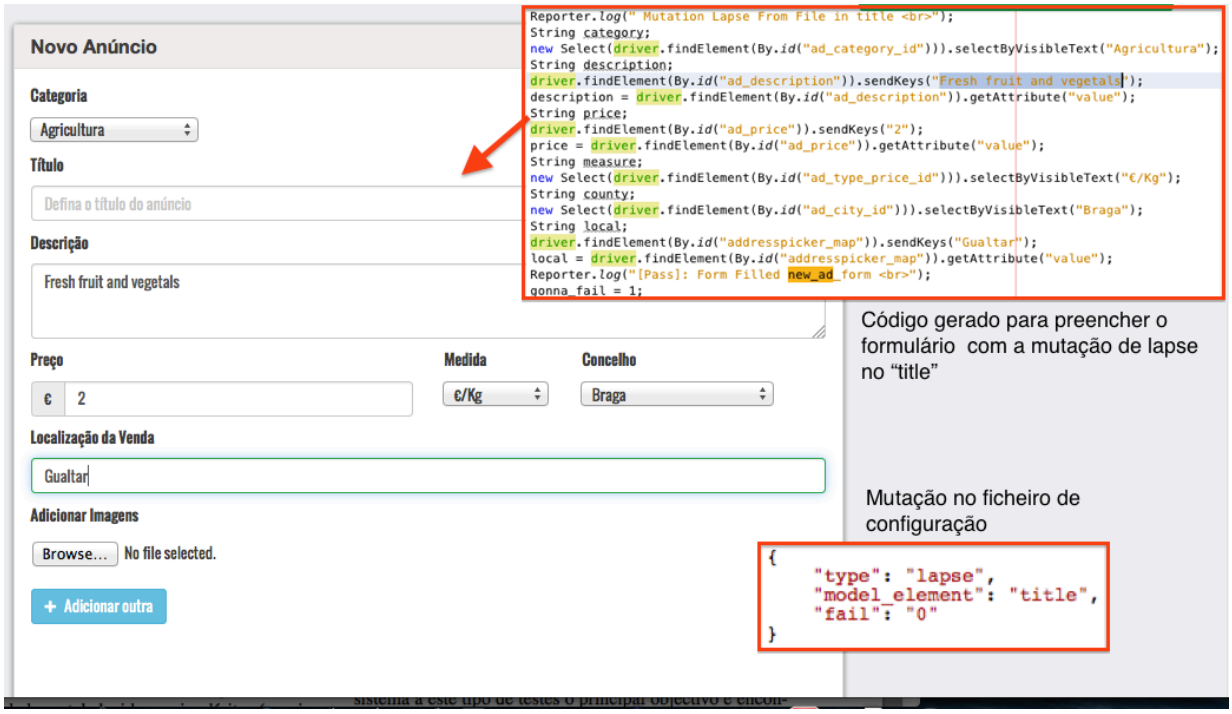

Figura 7.8: Exemplo de código gerado com uma mutação

o login/registo foram realizados com sucesso. Essa validação introduzida no modelo, de modo a que depois de sair do estado do login ou registo seria feita a confirmação que a notificação apareceria.

Depois de executar os testes e analisar o relatório, foi então possível detectar que um deles, o que verificava se a notificação era mostrada depois de fazer o *login*, tinha falhado. A Figura 7.10 apresenta o relatório do teste *path5_test4* onde é possível ver que o teste falhou com o erro *'Unable to locate element'*.

No modelo estava especificado que deveria aparecer uma notificação depois de realizar o login/registo. Portanto estava prevista no desenho da interface. Portanto o erro detectado é uma falha de implementação. Desta forma foi possível com a ferramenta detectar a falta de um elemento que daria um feedback ao utilizador.

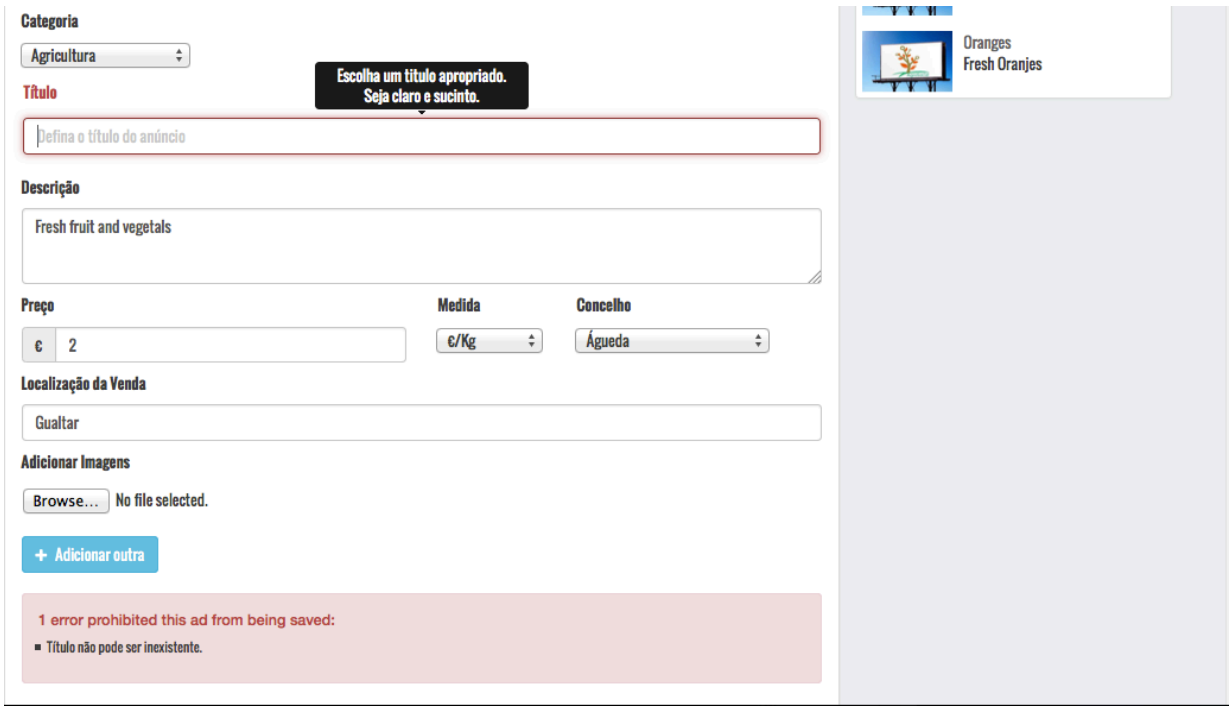

Figura 7.9: Resultado da submissão do formulário com mutações

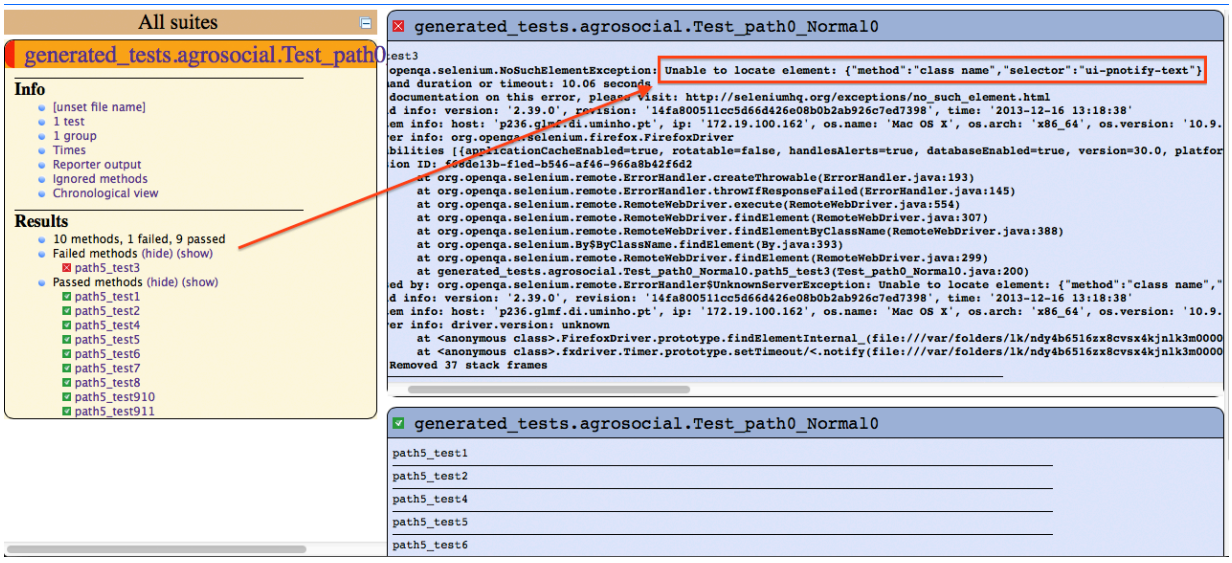

Figura 7.10: Relatório de teste

7.2 Twitter

Um segundo exemplo da aplicação da ferramenta será apresentado, neste caso o Twitter. Com este exemplo, pretende-se discutir um pouco mais sobre os números de caminhos encontrados e o numero de casos de teste gerados a partir do modelo criado.

Na Figura 7.12 é representada a máquina de estados construída a partir do modelo em SCXML (ver em anexo 23 modelo completo). Cada um dos estados representam uma página, menos os estados que se encontram dentro de outro estado, cujo significado será explicado abaixo. Podemos observar que tem um estado inicial chamado **root**. Este estado **root** representa a página de entrada do Twitter. Possui um formulário de *login* e um de registo e ainda um *link* para o *google play*. Cada um destes formulários estão representados como sub-estados do estado **root**, o *link* é representado por uma transição (*google_play_link*). As transições (representadas pelas setas) entre diferentes estados, podem ser por exemplo, *links* ou pedidos assíncronos ou outras acções que levem a novos estados.

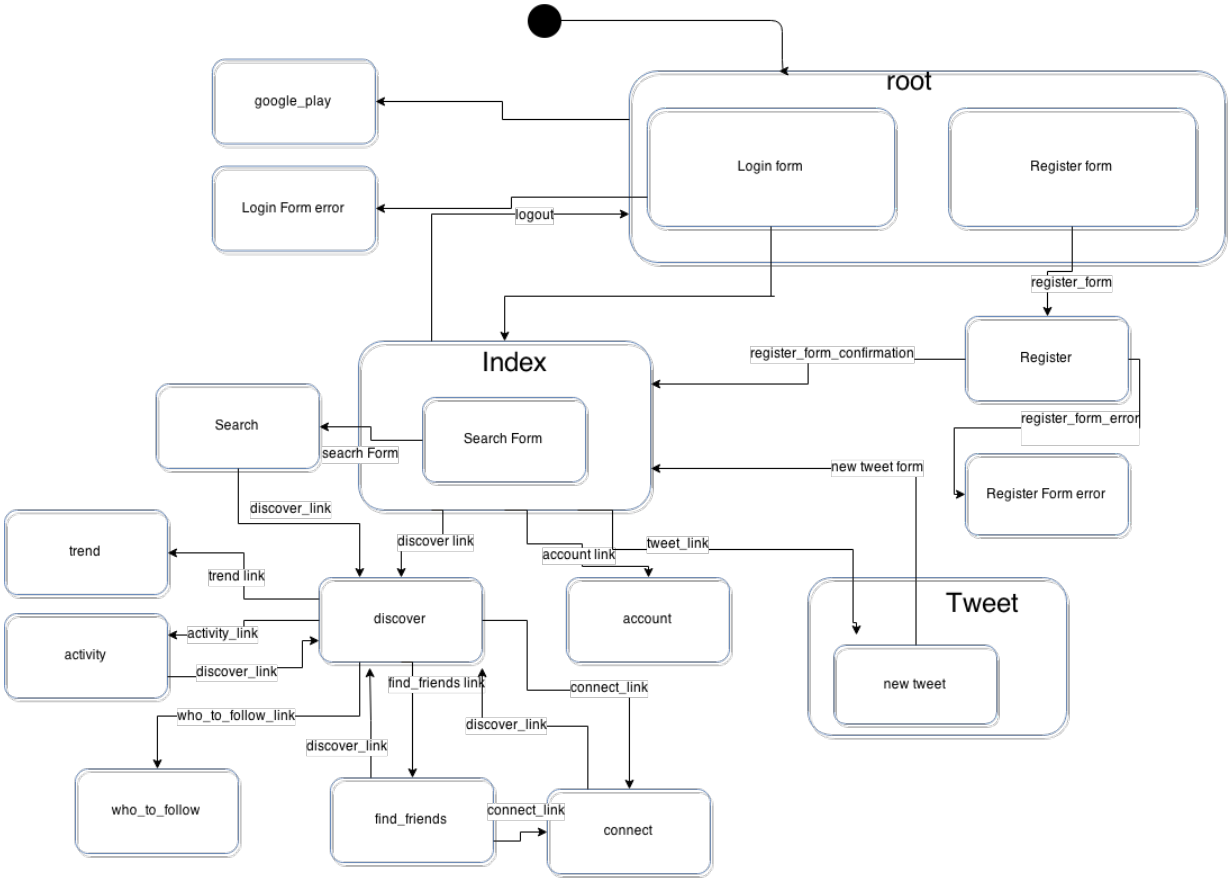

Figura 7.11: Representação da maquina de estados para o Twitter

7.2.1 Modelação de pedidos assíncronos

Podem existir dois tipos de pedido numa aplicação web, pedidos assíncronos e pedidos síncronos. Os pedidos assíncronos são uma forma eficiente de uma aplicação web tratar as interações com o utilizador com uma página web. Reduzindo a necessidade actualizar a página de forma completa. Nos pedidos síncronos há necessidade de actualizar a página após cada um deles.

Na página de **tweet** acontece um pedido assíncrono, em que ao submeter o formulário de novo tweet a página faz um pedido assíncrona não necessitando de ser feito um *refresh* à pagina. Nestes casos aquilo que se faz para se confirmar que a submissão realmente teve sucesso é adicionar uma validação sempre que se sai do estado de novo tweet, como por exemplo, verificar que o tweet acabado de inserir é mostrado depois de ser submetido.

7.2.2 Estatísticas

Foi construído um modelo com 15 estados onde pudessem ser gerados casos de teste com algumas das principais funcionalidades do sistema.

Com uma limite do número de vezes que um caminho pode passar por um nodo a ser 2 e um limite do numero de vezes que um caminho pode passar por uma certa aresta a ser 2, foram encontrados:

- 112 caminhos
- 1692 casos de teste

Com uma limite do número de vezes que um caminho pode passar por um nodo a ser 3 e um limite do numero de vezes que um caminho pode passar por uma certa aresta a ser 3, foram encontrados:

- 647 caminhos
- 14097 casos de teste

Para este caso em especifico foram gerados vários ficheiros de testes, em média cada ficheiro possui:

- 6000 linhas de código.
- 230 *asserts*.
- 150 métodos de teste.

Neste caso concreto a geração de casos de teste demorou tempos que a geração demorou **5.147 segundos**.

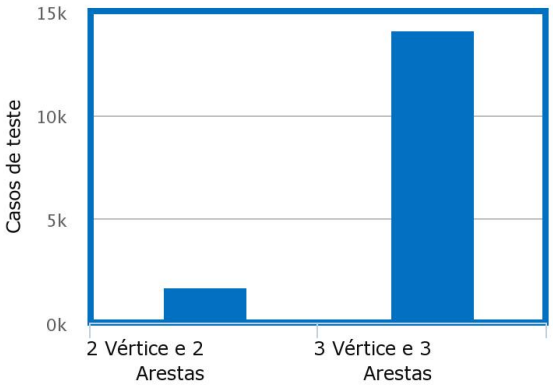

Figura 7.12: Casos de teste gerados para o Twitter com variações no algoritmo

7.2.3 Análise

Apesar de um modelo ser relativamente pequeno (apenas 15 estados), é possível perceber que se consegue gerar uma enorme quantidade de testes de forma automática de forma rápida. Obviamente que no futuro teriam de ser implementadas outras restrições no algoritmo para não gerar explosão de estados, realizar optimização nos testes encontrados de forma a minimizar as repetições de casos teste.

7.3 Conclusão

Neste capítulo foram apresentados dois exemplos de caso de estudo com a ferramenta desenvolvida. Foram demonstrados quais os passos necessários para testar uma funcionalidade do AgroSocial.

O principal objectivo era demonstrar o correcto funcionamento da ferramenta e de que era possível usá-la em vários tipos de aplicações web com pedidos síncronos e pedidos assíncronos.

8 | Conclusões

Neste capítulo será feita uma análise global ao trabalho realizado. Será também realizada uma proposta de trabalho futuro que posso ser desenvolvido.

Esta tese descreveu uma abordagem de testes baseados em modelos em aplicações web, que a partir de um modelo da interface gráfica verifica se o comportamento da aplicação se encontra de acordo com aquilo que foi definido nesse modelo. Conclui-se que o MBT pode ter um papel fundamental nos testes de software, garantindo uma maior qualidade de um produto de software. É praticamente inviável gerar a mesma carga de testes que é possível obter com MBT de forma manual. No MBT é essencial perceber quais as partes do SUT que são mais importantes e mais criticas para testar.

8.1 Objectivos Alcançados e Limitações

O objectivo principal era desenvolver uma ferramenta que a partir de um modelo gerasse os casos de testes. Foram testadas algumas aplicações, sendo geradas bastantes testes.

O maior custo desta abordagem acaba por ser na criação do modelo e na criação no ficheiro de mapeamento. No entanto o custo de manutenção acaba por ser bastante reduzido, e isso foi constatado quando eram necessárias alterações no modelo durante os casos de estudo realizados. Com esta abordagem foi possível gerar muitos casos de teste de forma automática, que não seria possível criar de forma manual. O mesmo número de casos de testes gerados com esta abordagem seria muito mais custoso se fosse gerado de forma manual. Outras das coisas feitas foram mudanças no modelo de certa forma para simular adição ou alteração de requisitos da aplicação. Aquilo que se constatou foi que o impacto dessas alterações eram pequenas, já que os testes eram novamente gerados de forma automática.

Outro dos objectivos era de realizar mutações nos testes para simular comportamentos atípicos por parte dos utilizadores. Todas as implementações referidas no documento foram implementadas com sucesso.

A nível de tecnologias as escolhas recaíram sobre JAVA, Selenium RC e sobre TESTNG. É possível concluir que a escolha das tecnologias foi a correcta. O Selenium RC permite cobrir praticamente todas a interacções possíveis de realizar sobre uma GUI.

Foi possível detectar alguns erros em interfaces gráfica utilizando a ferramenta, nomeadamente na aplicação do AgroSocial.

As principais funcionalidades da ferramenta são as seguintes:

- Gerar caminhos através dos grafos
- Gerar casos de teste com Selenium e TestNG
- Executar os testes no *browser*
- Gerar um relatório para cada um dos conjuntos de testes
- Verificar se *links*/imagens estão correctos
- Guardar uma imagem da interface da aplicação quando surge um erro
- Ser utilizável com diferentes *browsers* (Chrome e Firefox)
- Realizar validações (*is_displayed?, not_displayed?, enabled?, exits?, Css Properties,..*).
- Navegar em menus
- Interagir com *text fields*, *check boxes* e *select boxes* e grande parte do restantes elementos HTML.
- Realizar mutações aleatórias ou a possibilidade de escolher as quais mutações a realizar a partir de um ficheiro de configuração.
- Realizar operações *javascript* e assíncronas.

Apesar dos testes não serem de grande complexidade, aumentam a garantia da qualidade do *software*. Com a geração automáticas destes testes aumenta-se a confiabilidade do *software*, aumentando a confiança dos utilizadores.

8.2 Trabalho Futuro

Com a solução desenvolvida foi criada uma base para se ir construindo e melhorando a ferramenta. Foi uma solução desenhada e desenvolvida de raiz mas pensada e estruturada para evoluir. De forma a tornar a ferramenta mais robusta e completa foram pensadas nas seguintes funcionalidades para serem implementadas no futuro:

- Criação de uma ferramenta gráfica para a criação do modelo de forma a impedir o utilizador de cometer erros e de criar máquinas de estado inválidas.
- Geração semiautomática dos ficheiros de configuração.
- Melhorar os algoritmos de geração de caminhos para os casos de teste, de forma, a restringir mais
- Possibilidade de criação de mais tags para poder incorporar mais *tags*
- Integração com mais eventos de Javascript como, por exemplo, *drag and drop.*

Bibliografia

[1] V model waterfall model. http://www.waterfall-model.com/v-model-waterfall-model/.

[2] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[3] Kulvinder Singh, Semmi Department of Computer Science, and UIET Engineering. Finite state machine based testing of web applications. *International Journal of Software and Web Sciences (IJSWS)*.

[4] P. J. Cruz and J. C. Campos. Ambiente de geração, mutação e execução de casos de teste para aplicações web,. In *Atas da Conferncia Interação 2013*, pages 45–52. Universidade de Trás-os-Montes e Alto Douro, 2013.

[5] P. Bourque A. Abran, J. W. Moore and R. Dupuis. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE, 2004.

[6] C. Sandler G. J. Myers and T. Badgett. *The Art of Software Testing*. Wiley Publishing, 2005.

[7] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[8] Stephan Weißleder. *Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines*. PhD thesis, Universität zu Berlin, 2010.

[9] V. Rungta. Software testing life cycle stlc. http://www.guru99.com/software-testing-life-cycle.html.

[10] IEEE. Ieee standard glossary of software engineering terminology, 1990.

[11] Ana Cristina Ramada Paiva Pimenta. *Automated Specification-Based Testing of Graphical User Interfaces*. PhD thesis, Engineering Faculty of Porto University, Department of Electrical and Computer Engineering, 2007.

[12] Clayton Lewis, Peter Polson, Cathleen Wharton, and John Rieman. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *CHI '90 Proceedings*, pages 235–242. ACM Press, April 1990.

[13] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *CHI '90 Proceedings*, pages 249–256. ACM Press, April 1990.

[14] Watir webdriver. http://watirwebdriver.com.

[15] Selenium - web browser automation. http://docs.seleniumhq.org/docs/.

[16] Sikuli script - home. http://doc.sikuli.org.

[17] Atif M. Memon. *A Comprehensive Framework For Testing Graphical User Interfaces*. PhD thesis, 2001.

[18] Joonas Lindholm. Model-based testing. 2006.

[19] A Hartman. Adaptation of model based testing to industry (presentation slides). In *Agile and Automated Testing Seminar*, Tampere University of Technology, Tampere, Finland, August 2006.

[20] J. C. Campos J. L. Silva and A. C. R. Paiva. Model-based user interface testing with spec explorer and concurtasktrees. In *Electron. Notes Theor. Comput. Sci.*, volume 208, pages 77–93, 2007.

[21] Multimodal teresa - tool for design and development of multi-platform applications. http://giove.isti.cnr.it/teresa.html.

[22] J. T Reason. *Human error*. Cambridge : Cambridge University Press, 1990.

[23] F. Belli. Finite-state testing and analysis of graphical user interfaces. In *Proc. of the 12th ISSRE*. IEEE Computer Society Press, 2001.

[24] ATT Bell Laboratories Murray Hill, editor. *Principles and methods of testing finite state machines — A Survey*, New Jersey.

[25] A. Abdurazik J. Offutt, S. Y . Liu and P . Ammann. Generating test data from state-based specifications. *The Journal of Software Testing, Verification, and Reliability*, pages pp. 25–53, 2003.

[26] Nicola Migliorini. Comparing uml, scxml and the apache implementation of scxml. *European Southern Observatory Garching bei Munchen, Germany*.

[27] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[28] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[29] W3C. State chart xml (scxml) : State machine notation for control abstraction w3c working draft 16 february 2012, 2012.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Anexos

A | Casos de Estudo

A.1 Caso de estudo do Agrosocial

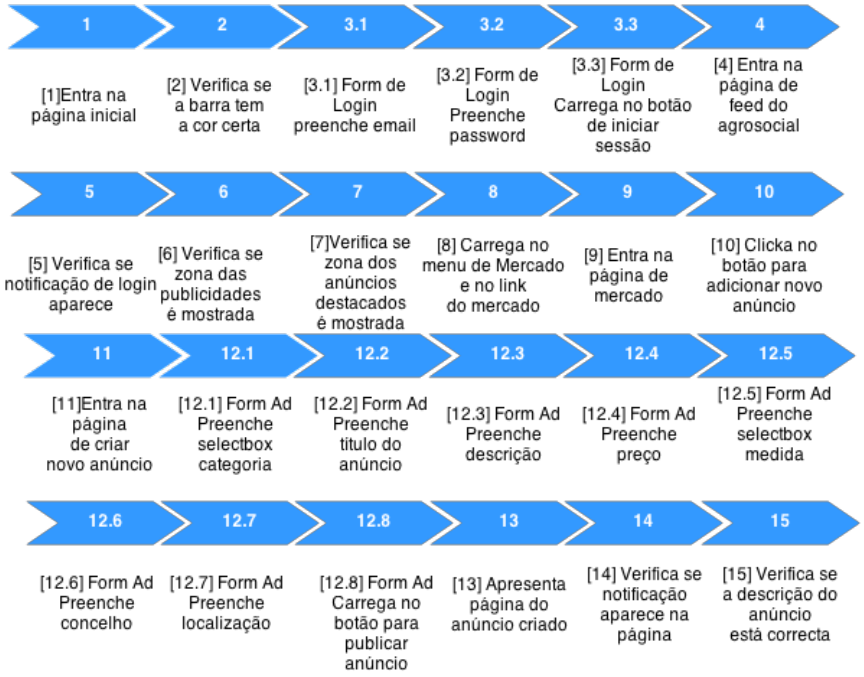

Figura A.1: Caminho abstracto encontrado

```
<scxml xmlns="http://www.w3.org/2005/07/scxml" initial="root">
 <state id="root">
  <state id="login_form" type="form" >
   <send label="email" type="required"/>
   <send label="password" type="required"/>
```

```
    <transition  type="form" label="submit_login_form" >

      <submit target="feed" />

      <error target="login_error" />

    </transition>

    <!--validation-->
    <onexit id="message_authentication" type="default"/>

  </state><!-- END OF LOGIN FORM -->


  <!-- LINKS -->
  <transition id="register" target="register"/>

  <!-- Validations onentry in state -->
  <onentry id="navbarAgro" type="css"/>

</state><!-- END OF ROOT PAGE-->


<state id="feed">

  <!-- LINKS -->
  <transition id="mercado" target="mercado" type="menu">
    <step  id="message_hover" target="mercado" />
    <step  id="menu_mercado" target="mercado" />
    <step   id="mercado" target="mercado" />
  </transition>

  <transition id="feed" target="feed" />

  <!-- LINKS -->
  <transition id="perfil" target="perfil" />


  <onentry id="publicidade" type="displayed?"/>
  <onentry id="destaques" type="displayed?"/>

</state>


<state id="mercado">

  <!-- LINKS -->
  <transition id="ad" target="ad" type="menu">
    <step  id="ad_hover" target="ad" />
    <step  id="ad" target="ad" />
```

```
  </transition>

  <transition id="perfil" target="perfil" type="menu">
    <step  id="menu_perfil" target="perfil" />
    <step   id="perfil" target="perfil" />
  </transition>

  <transition id="new_ad" target="new_ad"/>

  <state id="search_form" type="form" >

    <send label="city_search"/>

    <transition  type="form"  label="submit_search_form">
      <submit target="mercado1" />
    </transition>

  </state><!-- END OF LOGIN FORM -->



</state>



<state id="ad">

  <transition id="perfil" target="perfil" type="menu">
    <step  id="menu_perfil" target="perfil" />
    <step   id="perfil" target="perfil" />
  </transition>


  <state id="message_form" type="form" >

    <send label="message" type="required"/>

    <transition  type="form"  label="submit_message_form">

      <submit target="ad" />

    </transition>

  </state><!-- END OF LOGIN FORM -->

</state>


<state id="send_message">

</state>
```

```
<state id="perfil">

  <onentry id="avatar" type="displayed?"/>
</state>


<state id="new_ad">

  <state id="new_ad_form" type="form">

    <send label="category" type="required" element="selectbox"/>

    <send label="title" type="required"/>

    <send label="description" type="required"/>

    <send label="price" type="required" />

    <send label="measure" type="required" element="selectbox"/>

    <send label="county" type="required" element="selectbox"/>

    <send label="local" type="required"/>

    <transition  type="form"  label="submit_new_ad_form" >

      <submit target="ads" />

      <error target="login_error" />

    </transition>
    <!-- Verificar que anuncio tem o nome certo -->
    <!-- Verificar notificação -->
    <onexit id="notification_new_ad" type="displayed?"/>
    <onexit id="ads_title" type="displayed?"/>
    <onexit id="ads_expiration" type="displayed?"/>


  </state>

</state>


<state id="register">

  <state id="register_form" type="form" >

    <send label="user_name" type="required"/>

    <send label="email" type="required"/>
```

```
    <send label="password" type="required"/>
    <send label="password_confirmation" type="required"/>
    <send label="user_city_id" type="required" element="selectbox"/>
    <send label="user_birthday" type="required"/>
    <send label="user_phone" type="required"/>
    <send label="terms_and_conditions" type="required" element="checkbox"/>

    <transition  type="form"  label="submit_register_form" >
      <submit target="anuncio" />
      <error target="login_error" />
    </transition>

  </state><!-- END OF REGISTER FORM -->
 </state>

 <state id="login_error">
   <onentry id="error_message_login" type="displayed?"/>
 </state>

</scxml>
```

```
package generated_tests.agrosocial;

import java.io.File;
import java.util.List;
import org.apache.commons.io.FileUtils;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;
import org.openqa.selenium.OutputType;
import org.openqa.selenium.TakesScreenshot;
import org.openqa.selenium.firefox.FirefoxBinary;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.firefox.FirefoxProfile;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.openqa.selenium.support.ui.Select;
import org.openqa.selenium.interactions.Actions;
```

```
import org.openqa.selenium.WebElement;
import org.openqa.selenium.By;
import org.testng.Reporter;
import static org.testng.Assert.fail;
import static org.testng.Assert.assertTrue;
import java.util.concurrent.TimeUnit;
import java.io.IOException;
import java.lang.reflect.Method;
import org.junit.Ignore;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebDriverException;
import org.openqa.selenium.WebElement;
import org.testng.annotations.AfterGroups;
import org.testng.annotations.BeforeGroups;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

/**
 * Test Class for [root=>login_form, feed=>mercado, mercado=>search_form,
 * mercado1=>, root=>login_form, feed=>mercado, mercado=>ad, ad=>message_form,
 * ad=>, root=>login_form, feed=>mercado, mercado=>ad, ad=>perfil, perfil=>,
 * root=>login_form, feed=>mercado, mercado=>perfil, perfil=>, root=>login_form,
 * feed=>mercado, mercado=>new_ad, new_ad=>new_ad_form, ads=>, root=>login_form,
 * feed=>feed, feed=>, root=>login_form, feed=>perfil, perfil=>, root=>register,
 * register=>register_form, anuncio=>] Generated in 24/07/2014 15:05:42 Number
 * of test cases => 56 for 8 Groups(Found Paths)
 *
 */
public class Test_path0_Normal0 {

    private WebDriver driver;

    @BeforeGroups(groups = {
        "1",
        "2",
        "3",
        "4",
        "5",
        "6",
        "7",
        "8"
    })
    public void init_selenium() {
        FirefoxProfile profile = new FirefoxProfile();
        this.driver = new FirefoxDriver(new FirefoxBinary(new File("/Applications/Firefox.app/Contents/MacOS/firefox-bin")), profile);
        this.driver.get("http://localhost:3000/users/sign_in");
        this.driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
        WebDriverWait wait = new WebDriverWait(driver, 60);
    }

    @BeforeMethod
```

```
public void start_test(Method method) {
    String method_name;
    method_name = method.getName();
    Reporter.log("Started test " + method_name + "<br>");
}

@Ignore
public void my_wait()
        throws InterruptedException {
    Thread.sleep(3000);
}

private boolean isElementDisplayed(By by) {
    try {
        driver.findElement(by).isDisplayed();
        return true;
    } catch (AssertionError _x) {
        Reporter.log("Elemento nao econtrado");
        return false;
    }
}

@Ignore
private void brokenLinks() {
    List<WebElement> linkList = (driver.findElements(By.tagName("Links")));
    for (WebElement link : (linkList)) {
        try {
            HttpResponse response = new DefaultHttpClient().execute(new HttpGet(link.getAttribute("href")));
            if ((response.getStatusLine().getStatusCode() == 404)) {
                Reporter.log("[FAIL]:" + link.getAttribute("href"));
            }
        } catch (Exception _x) {
            Reporter.log("[FAIL]: " + link.getAttribute("href"));
        } finally {
            Reporter.log("[Pass]: All Links are good ");
        }
    }
}

@Ignore
private void brokenImages() {
    List<WebElement> linkList = (driver.findElements(By.tagName("Images")));
    for (WebElement link : (linkList)) {
        try {
            HttpResponse response = new DefaultHttpClient().execute(new HttpGet(link.getAttribute("src")));
            if ((response.getStatusLine().getStatusCode() == 404)) {
                Reporter.log("[FAIL]:" + link.getAttribute("src"));
            }
        } catch (Exception _x) {
            Reporter.log("[FAIL]: " + link.getAttribute("src"));
        } finally {
            Reporter.log("[Pass]: All Images are good ");
```

```
        }
    }
}

/**
 * Type of test => PATH => root=>login_form, feed=>mercado, mercado=>new_ad,
 * new_ad=>new_ad_form, ads=>
 *
 */
@Test(invocationCount = 1, groups = "5")
public void path5_test1()
        throws IOException, InterruptedException {
    int gonna_fail = 0;
    // page root
    Reporter.log(" Enter in page root <br>");
    // Asserts
    try {
        assertTrue(driver.findElement(By.id("navbarAgro")).getCssValue("background-color").equals("rgba(0, 167, 93, 1)"));
        Reporter.log("[Validaton]: Css Property is " + driver.findElement(By.id("navbarAgro")).getCssValue("background-color").equals("rgba(0, 167, 93, 1)<br>"));

    } catch (AssertionError _x) {
        Reporter.log("[Fail]: In a Validation navbarAgro ");
        fail(" [Fail]: In a Validation navbarAgro [Message] => " + _x.getMessage());
    }

    my_wait();
}

/**
 * Type of test => PATH => root=>login_form, feed=>mercado, mercado=>new_ad,
 * new_ad=>new_ad_form, ads=>
 *
 */
@Test(invocationCount = 1, groups = "5")
public void path5_test2()
        throws IOException, InterruptedException {
    int gonna_fail = 0;
    // Form login_form
    try {
            String email;
            driver.findElement(By.id("user_email")).sendKeys("raphaeljr28@gmail.com");
            email = driver.findElement(By.id("user_email")).getAttribute("value");
            String password;
            driver.findElement(By.id("user_password")).sendKeys("12345");
            password = driver.findElement(By.id("user_password")).getAttribute("value");
            Reporter.log("[Pass]: Form Filled login_form <br>");
            gonna_fail = 0;
    } catch (WebDriverException _x) {
        Reporter.log(" [FAIL]: Error in Form Fill submit_login_form " + _x.getMessage());
        fail(" [FAIL]: Error in Form Fill submit_login_form " + _x.getMessage());
    }
    if ((gonna_fail == 1)) {
```

```
        driver.findElement(By.cssSelector("input[name='commit']")).submit();
        Reporter.log("[Pass]: Click Submit Button  <br>");
        try {
            // Started validation for an error page
            assertTrue(driver.findElement(By.id("error_explanation")).isDisplayed());
            Reporter.log("Element displayed? " + driver.findElement(By.id("error_explanation")).isDisplayed());
        } catch (java.lang.AssertionError _x) {
            Reporter.log("[Negative Test Fail]: Error on Validations in an error state");
            fail("[Negative Test Fail]: Error on Validations in an error state " + _x.getMessage());
        }
        return;
    }
    driver.findElement(By.cssSelector("input[name='commit']")).submit();
    Reporter.log("[Pass]: Click Submit Button  <br>");
    Reporter.log("[Pass]: Form Completed  ID:login_form <br>");
    my_wait();
}

/**
 * Type of test => Validation After a Form PATH =>
 *
 */
@Test(invocationCount = 1, groups = "5")
public void path5_test3()
        throws IOException, InterruptedException {
    int gonna_fail = 0;
    try {
        assertTrue(driver.findElement(By.className("ui-pnotify-text")).getText().equals("Autenticação efetuada com sucesso."));
        Reporter.log("[Validaton]: Text is correct? " + driver.findElement(By.className("ui-pnotify-text")).getText().equals("Autenticação efetuada com sucesso."));

    } catch (AssertionError _x) {
        Reporter.log("[Fail]: In a Validation message_authentication after the form ");
        fail(" [Fail]: In a Validation [Message] => " + _x.getMessage());
    }
}

/**
 * Type of test => PATH => root=>login_form, feed=>mercado, mercado=>new_ad,
 * new_ad=>new_ad_form, ads=>
 *
 */
@Test(invocationCount = 1, groups = "5")
public void path5_test4()
        throws IOException, InterruptedException {
    int gonna_fail = 0;
    // page feed
    Reporter.log(" Enter in page feed <br>");
    // Asserts
    try {
        assertTrue(driver.findElement(By.cssSelector("div.col-lg-12:nth-child(3) > section:nth-child(1)")).isDisplayed());
        Reporter.log("Element displayed? " + driver.findElement(By.cssSelector("div.col-lg-12:nth-child(3) > section:nth-child(1)")).isDisplayed());
    } catch (AssertionError _x) {
```

```
            Reporter.log("[Fail]: In a Validation publicidade ");
            fail(" [Fail]: In a Validation publicidade [Message] => " + _x.getMessage());
        }

        my_wait();
    }

    /**
     * Type of test => Validation Test On entry in a Page PATH =>
     *
     */
    @Test(invocationCount = 1, groups = "5")
    public void path5_test5()
            throws IOException, InterruptedException {
        int gonna_fail = 0;
        try {
            assertTrue(driver.findElement(By.cssSelector("div.col-lg-12:nth-child(2) > section:nth-child(1)")).isDisplayed());
            Reporter.log("Element displayed? " + driver.findElement(By.cssSelector("div.col-lg-12:nth-child(2) > section:nth-child(1)")).isDisplayed());
        } catch (AssertionError _x) {
            Reporter.log("[Fail]: In a Validation destaques ");
            fail(" [Fail]: In a Validation [Message] => " + _x.getMessage());
        }
    }

    /**
     * Type of test => PATH => root=>login_form, feed=>mercado, mercado=>new_ad,
     * new_ad=>new_ad_form, ads=>
     *
     */
    @Test(invocationCount = 1, groups = "5")
    public void path5_test6()
            throws IOException, InterruptedException {
        int gonna_fail = 0;
        // click mercado
        try {
                Actions actions = new Actions(driver);
                WebElement Mainmenu = driver.findElement(By.cssSelector("div.panel:nth-child(2) > div:nth-child(1)"));
                actions.click(Mainmenu).build().perform();
                Reporter.log("[Pass]: Click in a Menu message_hover <br>");
        } catch (WebDriverException _x) {
            Reporter.log(" Error in Menu Click message_hover " + _x.getMessage());
            fail(" Error in Menu Click message_hover " + _x.getMessage());
        }
        try {
            driver.findElement(By.xpath("//div[@id='accordion']/div[2]/div/h4/a")).click();
        } catch (WebDriverException _x) {
            Reporter.log(" [FAIL]: Error in Menu Click " + _x.getMessage());
        }
        driver.findElement(By.linkText("Entrar no Mercado")).click();
        my_wait();
        Reporter.log("[Pass]: Call Completed ID:mercado <br>");
        my_wait();
```

```
}

/**
 * Type of test => PATH => root=>login_form, feed=>mercado, mercado=>new_ad,
 * new_ad=>new_ad_form, ads=>
 *
 */
@Test(invocationCount = 1, groups = "5")
public void path5_test7()
        throws IOException, InterruptedException {
    int gonna_fail = 0;
    // click new_ad
    try {
            driver.findElement(By.cssSelector("#breadcrumbAds > a.url")).click();
    } catch (WebDriverException _x) {
        Reporter.log(" [FAIL]: Error in Call new_ad " + _x.getMessage());
        fail(" [FAIL]: Error in Call new_ad " + _x.getMessage());
    }
    Reporter.log("[Pass]: Call Completed ID:new_ad <br>");
    my_wait();
}

/**
 * Type of test => PATH => root=>login_form, feed=>mercado, mercado=>new_ad,
 * new_ad=>new_ad_form, ads=>
 *
 */
@Test(invocationCount = 1, groups = "5")
public void path5_test8()
        throws IOException, InterruptedException {
    int gonna_fail = 0;
    // page new_ad
    Reporter.log(" Enter in page new_ad <br>");
    // Asserts
    try {
        assertTrue(driver.findElement(By.xpath("//*[@id='ad-form']/div[6]/div[2]")).isDisplayed());
        Reporter.log("Element displayed? " + driver.findElement(By.xpath("//*[@id='ad-form']/div[6]/div[2]")).isDisplayed());
    } catch (AssertionError _x) {
        Reporter.log("[Fail]: In a Validation googlemaps_new_ad ");
        fail(" [Fail]: In a Validation googlemaps_new_ad [Message] => " + _x.getMessage());
    }

    my_wait();
}

/**
 * Type of test => PATH => root=>login_form, feed=>mercado, mercado=>new_ad,
 * new_ad=>new_ad_form, ads=>
 *
 */
@Test(invocationCount = 1, groups = "5")
public void path5_test9()
```

```
        throws IOException, InterruptedException {
    int gonna_fail = 0;
    // Form new_ad_form
    try {
            String category;
            new Select(driver.findElement(By.id("ad_category_id"))).selectByVisibleText("Agricultura");
            String title;
            driver.findElement(By.id("ad_title")).sendKeys("Fruits and Vegetals");
            title = driver.findElement(By.id("ad_title")).getAttribute("value");
            String description;
            driver.findElement(By.id("ad_description")).sendKeys("Fresh fruit and vegetals");
            description = driver.findElement(By.id("ad_description")).getAttribute("value");
            String price;
            driver.findElement(By.id("ad_price")).sendKeys("2");
            price = driver.findElement(By.id("ad_price")).getAttribute("value");
            String measure;
            new Select(driver.findElement(By.id("ad_type_price_id"))).selectByVisibleText("€/Kg");
            String county;
            new Select(driver.findElement(By.id("ad_city_id"))).selectByVisibleText("Braga");
            String local;
            driver.findElement(By.id("addresspicker_map")).sendKeys("Gualtar");
            local = driver.findElement(By.id("addresspicker_map")).getAttribute("value");
            Reporter.log("[Pass]: Form Filled new_ad_form <br>");
            gonna_fail = 0;
    } catch (WebDriverException _x) {
        Reporter.log(" [FAIL]: Error in Form Fill submit_new_ad_form " + _x.getMessage());
        fail(" [FAIL]: Error in Form Fill submit_new_ad_form " + _x.getMessage());
    }
    if ((gonna_fail == 1)) {
        driver.findElement(By.cssSelector("div.actions > input[name='commit']")).submit();
        Reporter.log("[Pass]: Click Submit Button  <br>");
        try {
            // Started validation for an error page
            assertTrue(driver.findElement(By.id("error_explanation")).isDisplayed());
            Reporter.log("Element displayed? " + driver.findElement(By.id("error_explanation")).isDisplayed());
        } catch (java.lang.AssertionError _x) {
            Reporter.log("[Negative Test Fail]: Error on Validations in an error state");
            fail("[Negative Test Fail]: Error on Validations in an error state " + _x.getMessage());
        }
        return;
    }
    driver.findElement(By.cssSelector("div.actions > input[name='commit']")).submit();
    Reporter.log("[Pass]: Click Submit Button  <br>");
    Reporter.log("[Pass]: Form Completed  ID:new_ad_form <br>");
    my_wait();
}

/**
 * Type of test => Validation After a Form PATH =>
 *
 */
@Test(invocationCount = 1, groups = "5")
```

```
public void path5_test10()
        throws IOException, InterruptedException {
    int gonna_fail = 0;
    try {
        assertTrue(driver.findElement(By.cssSelector("ads_title")).isDisplayed());
        Reporter.log("Element displayed? " + driver.findElement(By.cssSelector("ads_title")).isDisplayed());
    } catch (AssertionError _x) {
        Reporter.log("[Fail]: In a Validation ads_title after the form ");
        fail(" [Fail]: In a Validation [Message] => " + _x.getMessage());
    }
}

/**
 * Type of test => Validation After a Form PATH =>
 *
 */
@Test(invocationCount = 1, groups = "5")
public void path5_test11()
        throws IOException, InterruptedException {
    int gonna_fail = 0;
    try {
        assertTrue(driver.findElement(By.cssSelector(".ads_description > div:nth-child(5) > div:nth-child(1)")).isDisplayed());
        Reporter.log("Element displayed? " + driver.findElement(By.cssSelector(".ads_description > div:nth-child(5) > div:nth-child(1)")).isDisplayed());
    } catch (AssertionError _x) {
        Reporter.log("[Fail]: In a Validation ads_expiration after the form ");
        fail(" [Fail]: In a Validation [Message] => " + _x.getMessage());
    }
}

/**
 * Type of test => Validation After a Form PATH =>
 *
 */
@Test(invocationCount = 1, groups = "5")
public void path5_test12()
        throws IOException, InterruptedException {
    int gonna_fail = 0;
    try {
        assertTrue(driver.findElement(By.cssSelector("ui-pnotify-text")).isDisplayed());
        Reporter.log("Element displayed? " + driver.findElement(By.cssSelector("ui-pnotify-text")).isDisplayed());
    } catch (AssertionError _x) {
        Reporter.log("[Fail]: In a Validation notification_new_ad after the form ");
        fail(" [Fail]: In a Validation [Message] => " + _x.getMessage());
    }
}

@AfterGroups(groups = {
    "1",
    "2",
    "3",
    "4",
    "5",
```

```
        "6",
        "7",
        "8"
    })
    public void closeBrowsers() {
        this.driver.close();
    }

}
```

A.2 Caso de estudo do Twitter

```
<scxml xmlns="http://www.w3.org/2005/07/scxml" initial="root">

  <state id="root">

    <state id="login_form" type="form" >

      <send label="email" type="required"/>

      <send label="password" type="required"/>


      <transition type="form" label="submit_login_form">

        <submit target="index" />

        <error target="error_login" />

      </transition>

      <onexit id="icon_header" type="displayed?"/>

    </state><!-- END OF LOGIN FORM -->


    <state id="register_form" type="form" >

      <send label="name_reg" type="required"/>

      <send label="email_reg" type="required"/>

      <send label="password_reg" type="required"/>


      <transition type="form" label="submit_register_form">

        <submit target="register" />
```

```
      <error target="error_register" />

    </transition>

     <onexit id="icon_header" type="displayed?"/>


  </state>


<onentry id="icon_header" type="displayed?"/>
<onentry id="cookie_message" type="contains"/>


</state><!-- END OF ROOT PAGE-->

<!-- ************************ STATE ****************************** -->
<state id="index">

 <state id="search" type="form" >

  <send label="search_query" type="required"/>

  <transition type="AJAX" label="submit_seacrh_form" >
   <submit target="search" />
  </transition>

 </state><!-- END OF LOGIN FORM -->


 <transition type="menu">
  <step id="settings_menu" target="root" />
  <step id="settings_menu" target="root" />
  <step id="logout" target="root" />
 </transition>

 <transition id="discover" target="discover"/>
 <transition id="account" target="account"/>
 <transition id="tweet" target="tweet"/>

 <onentry id="image_profile" type="displayed?"/>
 <onentry id="twitter_icon" type="displayed?"/>
 <onentry id="name_profile" type="contains"/>

</state><!-- END OF INDEX PAGE-->


<!-- ************************ STATE ****************************** -->
<state id="tweet">

 <state id="new_tweet" type="form" >
```

```
    <send label="tweet_box" type="required"/>

    <transition type="AJAX" label="submit_new_tweet_form"  >

      <submit target="tweet" />

    </transition>

  </state><!-- END OF LOGIN FORM -->

  <transition id="close_tweet" target="close_tweet"/>

  <onentry id="submit_new_tweet_form" type="disabled?"/>

</state><!-- END OF INDEX PAGE-->

<!-- ************************  STATE ******************************** -->
<state id="discover">

  <!-- LINKS -->
  <transition id="connect" target="connect"/>
  <transition id="find_friends" target="find_friends"/>
  <transition id="activity" target="activity"/>
  <transition id="who_to_follow" target="who_to_follow"/>

  <transition id="trend" target="trend"/>


  <onentry id="content-header" type="contains"/>
  <onentry id="trends" type="displayed?"/>

</state><!-- END OF INDEX PAGE-->


<!-- ************************  STATE ******************************** -->
<state id="trend">

    <onentry id="content-main-heading" type="contains"/>
</state><!-- END OF INDEX PAGE-->

<!-- ************************  STATE ******************************** -->
<state id="account">

</state><!-- END OF INDEX PAGE-->

<!-- ************************  STATE ******************************** -->
<state id="search">
  <transition id="discover" target="discover"/>

  <onentry id="content-main-heading" type="default"/>
  <onentry id="image_profile" type="displayed?"/>
```

```
</state>


<!-- ************************ STATE ****************************** -->
<state id="who_to_follow">

  <onentry id="header-inner" type="contains"/>

</state>
<!-- ************************ STATE ****************************** -->
<state id="google_play">

</state>

<!-- ************************ STATE ****************************** -->
<state id="activity">
  <transition id="discover" target="discover"/>
</state>

<!-- ************************ STATE ****************************** -->
<state id="connect">
  <transition id="discover" target="discover"/>
</state>

<!-- ************************ STATE ****************************** -->
<state id="find_friends">

  <transition id="discover" target="discover"/>
  <transition id="connect" target="connect"/>
</state>

<!-- ************************ STATE ****************************** -->
<state id="logout">

</state>

<!-- ************************ STATE ****************************** -->
<state id="register">

  <onentry id="error_message" type="default"/>
</state>

<!-- ************************ STATE ****************************** -->
<state id="error_register">
  <onentry id="error_message" type="default"/>
</state>


<!-- ************************ STATE ****************************** -->
<state id="error_login">
  <onentry id="error_message" type="default"/>
</state>
```

```
</scxml>
```

Listagem 23: Código gerado para o twitter