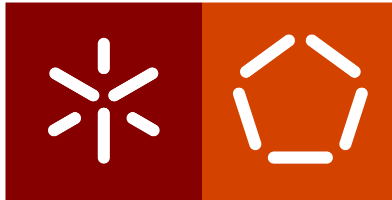
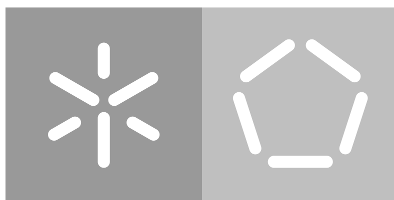


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Miguel Carvalho Pinto

TOM Framework: Uma ferramenta de testes baseados em modelos para interfaces gráficas web

Março de 2017



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Miguel Carvalho Pinto

TOM Framework: Uma ferramenta de testes baseados em modelos para interfaces gráficas web

Dissertação de mestrado
Mestrado em Engenharia Informática

Dissertação supervisionada por
José Creissac Campos

Março de 2017

AGRADECIMENTOS

Gostaria de agradecer, em especial, ao meu orientador, Professor José Creissac Campos, pela oportunidade, apoio e sugestões de desenvolvimento desta dissertação.

Aos meus pais, às minhas irmãs, à minha namorada e aos meus amigos um obrigado pelo incentivo e apoio incondicional dado.

A todos aqueles que estiverem presentes no decorrer da minha formação, o meu obrigado.

ABSTRACT

The techniques of Model Based Testing (MBT) compare the behaviours of the system under test with the system model (the oracle). The application of MBT to the Graphical User Interface allows a more exaustive and continuous evaluation of the system, through the simulation of user actions on the graphical interface. As such, it is possible to meaning substantially reduce the cost of system evaluation and, eventually, identify implementation errors through the GUI, without the involvement of external users. This process occurs through the execution of test cases, generated from the system model, on the application under test. These are the tests case that verify if the implementation is following the model, ensuring the improvement of the developed system's quality.

This dissertation describes a MBT tool for web applications, the TOM Framework. Part of the framework (TOM Generator) takes advantage of previously developed work, the other (TOM editor) is presented here. The main goals of the framework are to automate and facilitate the creation of models of the system that will be used to automatically generate executable test cases in the graphical interface under test. The capture and interpretation of user interaction with the web application under test was one of the challenges that was overcome during the development of this dissertation. At the end of it, one can find an application of the framework in a case study.

RESUMO

As técnicas de teste baseados em modelos (do inglês, *Model Based Testing (MBT)*) comparam o comportamento do sistema sob teste com o comportamento do modelo do sistema (o oráculo). A aplicação de MBT às interfaces gráficas do utilizador (do inglês, *Graphical User Interface (GUI)*) permite uma avaliação mais exaustiva e contínua do sistema, através da simulação de ações do utilizador com a interface gráfica. Desta forma, é possível reduzir significativamente o custo de avaliação do sistema, e identificar, eventualmente, erros de implementação através da GUI, sem o envolvimento de utilizadores externos. Este processo decorre através da execução dos casos de teste, gerados a partir do modelo do sistema, na aplicação sobre teste. São estes casos de teste que verificam se a implementação está de acordo com o modelo, assegurando assim uma melhoria da qualidade do sistema desenvolvido.

Esta dissertação descreve uma ferramenta de MBT para aplicações web, a TOM Framework. Parte da *framework* (TOM Generator) aproveita trabalho anteriormente desenvolvido, a outra (TOM Editor) é aqui apresentada. Os objetivos principais da *framework* passam por automatizar e facilitar a criação de modelos do sistema que, posteriormente, são utilizados para gerar automaticamente casos de teste executavéis na interface gráfica sobre teste. A captura e interpretação da interação do utilizador com a aplicação web sobre teste foi um dos desafios ultrapassados no desenvolvimento desta dissertação. No final da mesma, encontra-se uma aplicação da framework a um caso de estudo.

CONTEÚDO

1 INTRODUÇÃO 1
- 1.1 Descrição do projeto 2
- 1.2 Estrutura do documento 3

2 INTERFACES DO UTILIZADOR 4
- 2.1 Interação com a UI do sistema 4
- 2.2 Tipos de interfaces do utilizador 5
  - 2.2.1 Interface da linha de comandos 5
  - 2.2.2 Interface gráfica do utilizador 6
- 2.3 Evolução das interfaces gráficas do utilizador 6
- 2.4 Avaliação da usabilidade 8
  - 2.4.1 Métodos empíricos 8
  - 2.4.2 Métodos analíticos 10
- 2.5 Avaliação da qualidade de implementação 12
- 2.6 Testes baseados em modelos para interfaces gráficas 12
  - 2.6.1 Processo de MBT 13
  - 2.6.2 Benefícios e dificuldades 14
  - 2.6.3 Trabalhos relacionados 15
  - 2.6.4 Tecnologias relacionadas 16
- 2.7 Conclusões 16

3 TOM FRAMEWORK 18
- 3.1 Modelação do sistema 19
  - 3.1.1 Elementos principais da máquina de estados 19
  - 3.1.2 Atributos dos elementos da máquina de estados 21
- 3.2 Mapeamento e dados 23
- 3.3 Validações 26
- 3.4 Mutações 28
  - 3.4.1 Modelação das mutações no sistema 29
- 3.5 Configurações para a geração de casos de teste 31
- 3.6 Geração de casos de teste 32
- 3.7 Conclusões 34

4 IRIT: GERAÇÃO DE CENÁRIOS 35
- 4.1 Caso de estudo 35
- 4.2 Arquitetura 37

4.3 Conversão do modelo de tarefas 38
4.4 Geração dos cenários 40
4.5 Conclusões 41
5 TOM EDITOR 42
5.1 Abordagem proposta 42
5.2 Arquitetura da solução 43
5.3 Tecnologias e limitações 45
5.4 Comunicação entre aplicação e o browser 48
5.5 Funcionalidades da extensão 50
5.5.1 Selecção do elemento na DOM 52
5.6 Captura da interação do utilizador 53
5.7 Exportação de dados 55
5.8 Conclusões 56
6 APLICAÇÃO DA TOM FRAMEWORK 57
6.1 Modelo do sistema 57
6.2 Configurações no TOM Generator 62
6.3 Geração e execução dos casos de teste 64
6.4 Resultados e discussão 66
6.5 Conclusões 70
7 CONCLUSÕES E TRABALHO FUTURO 71
7.1 Trabalho futuro 72

Bibliografia 74

A MODELOS DO SISTEMA 78
A.1 OntoWorks 78
A.1.1 Máquina de estados 78
A.1.2 Mapeamento 81
A.1.3 Valores 92
A.1.4 Mutações 93

LISTA DE FIGURAS

Figura 1 As etapas do ciclo de interação (Norman, 1988). 5
Figura 2 *Screenshot* da interface Star (Raymond and Landley, 2004). 7
Figura 3 Processo de *Model Based Testing* (Utting and Legeard, 2007). 14
Figura 4 Processo de MBT do TOM Generator 19
Figura 5 Modelo de tarefas HAMSTERS para a tarefa prinicipal *Check for thunderstorm and avoid them if necessary*. 36
Figura 6 Painel de controlo EFIS, com b) e sem a) ativação do radar metereológico. 37
Figura 7 Arquitetura da geração de cenários 38
Figura 8 *Screenshot* da ferramenta HAMSTERS em simulação. 39
Figura 9 Modelo de domínio do TOM Editor 43
Figura 10 Mockups da interface do TOM Editor 44
Figura 11 Interface gráfica da janela de popup das extensões no Google Chrome. 45
Figura 12 Interface gráfica da página de aplicações do Google Chrome. 46
Figura 13 Interface gráfica da janela de inspeção do browser no Google Chrome. 46
Figura 14 Comunicação por componentes entre o TOM Editor e o *browser*. 49
Figura 15 Menus de adição de ações e estados do TOM Editor. 51
Figura 16 Formulário de adição de validações do TOM Editor. 51
Figura 17 Exemplo de seleção do texto na página web. 52
Figura 18 Mapeamento de formulário alterado, após seleção direta de elemento na página web. 53
Figura 19 Interface gráfica para a ativação do modo de captura de utilização. 54
Figura 20 Interface gráfica com os tipos de configurações exportáveis. 55
Figura 21 Funcionalidades disponíveis a partir do TOM Editor. 56
Figura 22 Interface gráfica com parte do modelo do sistema construído. 58
Figura 23 Interface gráfica da barra de navegação do OntoWorks. 59
Figura 24 Formulário de criação de uma validação e seleção de elemento na página web. 59
Figura 25 Interface gráfica com as validações do estado ”home”. 60
Figura 26 Interface gráfica com as definições das mutações. 61
Figura 27 Estado de erro (*sign in error*) com a validação existente. 61

Figura 28 Máquina de estados do OntoWorks. 62
Figura 29 Ficheiros de teste automaticamente gerados. 66
Figura 30 Resultados da execução dos casos de teste ao OntoWorks. 66
Figura 31 Problemas encontrados no caminho dos testes. 67
Figura 32 Comparação dos elementos HTML dos formulários de registro e *login* da aplicação. 68

LISTA DE TABELAS

Tabela 1 Tabela com os atributos do elemento *state*. 21

Tabela 2 Tabela com os atributos do elemento *transition*. 22

Tabela 3 Tabela com os atributos do elemento *transition* nos formulários. 22

Tabela 4 Tabela com os atributos do elemento *send*. 22

Tabela 5 Tabela com os atributos dos elementos *onentry* e *onexit*. 23

Tabela 6 Tabela com a descrição dos elementos de mapeamento do modelo. 24

Tabela 7 Tabela descritiva da estrutura do ficheiro de valores. 26

Tabela 8 Tabela com a descrição dos elementos de mutação. 30

Tabela 9 Tabela com os resultados da execução dos casos de teste no OntoWorks. 67

LISTA DE LISTAGENS

3.1 Exemplo da estrutura dos elementos. 21
3.2 Exemplo de modelação dos atributos. 23
3.3 Exemplo de mapeamentos dos elementos. 25
3.4 Exemplo do ficheiro de valores. 26
3.5 Exemplo de modelação da máquina de estados 28
3.6 Exemplo de configuração das mutações em ficheiro. 30
3.7 Configuração das variáveis do TOM Generator. 31
3.8 Exemplo da estrutura do código executável gerado. 33

4.1 Exemplo da máquina de estados convertida. 39
4.2 Parte do cenário gerado. 40

5.1 Formato da estrutura da mensagem de comunicação. 50

6.1 Exemplo de configuração do TOM Generator para a aplicação OntoWorks. . 63
6.2 Código gerado para o preenchimento do formulário de *Login*. 65
6.3 Exemplo de código gerado para a mutação *lapse*. 69

SIGLAS

A

API Application Programming Interface.

C

CLI Command Line Interface.

CSS Cascading Style Sheet.

D

DOM Document Object Mode.

DSL Domain Specification Language.

F

FCU Flight Control Unit.

G

GUI Graphical User Interface.

H

HTML Hypertext Markup Language.

I

IRIT Institut de Recherche en Informatique de Toulouse.

J

JS JavaScript.

JSON JavaScript Object Notation.

M

MBT Model Based Testing.

MVC Model-View Controller.

P

PBGT Pattern Based GUI Testing.

S

SCXML State Chart XML.

SUT System Under Test.

T

TS Typescript.

U

UI User Interface.

1

INTRODUÇÃO

A tecnologia evolui constantemente com novos serviços e aplicações, mais robustos e complexos, que asseguram a ligação das pessoas com o mundo. É essencial que nestes novos serviços não existam pontos de falha que possam causar erros ou possibilidades de acidentes no sistema. A indústria de software tem investido cada vez mais recursos de forma a validar e testar todo o seu trabalho antes da divulgação para o público. É então importante que existam ferramentas e soluções que permitam às empresas diminuir os custos, mas ao mesmo tempo acelerar o processo de execução de testes aos seus sistemas.

O teste de software é um processo que permite avaliar se um produto cumpre determinadas especificações e se funciona de acordo com o previsto. Tem como objetivo a detecção de falhas e/ou comportamentos indesejados, de forma a que estes problemas sejam corrigidos antes do final da fase de desenvolvimento. Os testes realizados à camada de apresentação das aplicações servem para verificar uma parte bastante importante do produto, a ligação das funcionalidades disponíveis do sistema com o utilizador e a resposta do sistema às interações do utilizador.

Uma interface do utilizador (do inglês, *User Interface (UI)*) deve estar preparada para que um utilizador seja capaz de atingir um determinado objetivo de forma eficiente e eficaz, determinando assim a usabilidade do sistema (Cruz and Campos, 2013). As dimensões da usabilidade que descrevem a qualidade de uma interface são descritas na norma ISO9241-11 (International Organization for Standardization, 1998). A eficácia define a precisão e integridade com que os utilizadores atingem os objetivos, a eficiência está relacionada com os recursos gastos ao atingir os objetivos e a satisfação com a reação positiva em relação à utilização do produto. Estas dimensões são tipicamente avaliadas em processos manuais de teste com utilizadores e/ou peritos.

As técnicas de teste baseados em modelos (do inglês, *Model Based Testing (MBT)*) permitem automatizar a geração de casos de teste a partir do modelo de sistema que está a ser testado (Barbosa et al., 2011). Através deste processo e com a utilização de modelos que descrevem a interação de um utilizador com uma interface gráfica do utilizador (do inglês,

Graphical User Interface (GUI)), é possível realizar a geração de casos de teste que testam se a interface funciona de acordo com os requisitos especificados, isto é, se os resultados finais estão de acordo com o comportamento esperado.

As GUI podem ser expressas de várias formas e estilos, em diferentes dimensões e nos mais variados tipos de dispositivos. São necessárias ferramentas que permitam testar as interfaces gráficas das aplicações dentro do ambiente em que elas foram construídas, sejam elas aplicações móveis, websites, ou aplicações *desktop*.

O TOM Generator é uma aplicação desenvolvida no laboratório HASLab de investigação em Software Confiável, que tem como principais características a geração e execução de casos de teste a interface gráficas Web com base num modelo da interface. Os casos de teste inicialmente gerados são independentes da linguagem e da tecnologia (e, por isso, denominados de casos de teste abstratos). Após a geração dos testes, podem ser introduzidas mutações nos casos de teste. Estas têm por objetivo simular pequenos erros de utilização na aplicação, obtendo-se novos casos de teste abstratos. Posteriormente, os casos de teste abstratos são transformados em casos de teste executáveis (isto é, executáveis), que estão definidos por uma linguagem e tecnologia.

1.1 DESCRIÇÃO DO PROJETO

Esta dissertação tem como objetivo principal o desenvolvimento de um *plugin* que permita facilitar e automatizar o processo de criação do modelo do sistema da interface gráfica que se pretende avaliar.

Atualmente, o processo de geração dos casos de teste no TOM Generator é baseado num modelo do sistema, que se encontra dividido em quatro ficheiros:

MÁQUINA DE ESTADOS
Representa o comportamento da interface gráfica.

MAPEAMENTOS
Representa a ligação entre a página web e os elementos da máquina de estados.

VALORES
Contém os dados para os testes. Por exemplo, os valores que devem ser enviados para os campos dos formulários.

MUTAÇÕES
Contém as definições das mutações escolhidas.

Acontece que, estes ficheiros são criados e definidos manualmente pelo utilizador com base na análise que o mesmo faz da aplicação sob teste, podendo originar facilmente erros

ao nível do mapeamento entre a interface da aplicação e o modelo. Além disso, à medida que se vai adicionando informação sobre o sistema aos ficheiros estes ficam menos claros, causando algumas dificuldades na edição dos ficheiros e um tempo de criação superior ao desejável.

Com a construção deste novo *plugin*, o TOM Editor, pretende-se diminuir os erros e o tempo dispendido a criar o modelo do sistema da GUI, automatizando e organizando os vários processos que o definem. Outra das características que se pretende adicionar a este *plugin* é a capacidade de o utilizador visualizar, instantaneamente, a forma de como o modelo se vai construindo, como se tratasse de um editor, fornecendo também funcionalidades de extração da informação para ficheiros, edição e remoção de dados.

Tendo em conta os aspectos anteriormente mencionados, os objetivos principais desta dissertação são:

1. Construção de um *plugin* que suporte a elaboração de modelos do sistema.
2. Facilitar o mapeamento entre a máquina de estados e a interface gráfica.
3. Automatizar o processo de construção da máquina de estados.

1.2 ESTRUTURA DO DOCUMENTO

A dissertação encontra-se organizada em 7 capítulos:

1. **Introdução** - Apresenta uma visão geral do projeto e do problema que se pretende solucionar.
2. **Interfaces do utilizador** - Aborda alguns aspectos das interfaces do utilizador, como a interação do utilizador e os tipos de avaliação das interfaces.
3. **TOM Framework** - Apresenta as funcionalidades e características da *framework*, em especial do TOM Generator.
4. **IRIT: Geração de cenários** - Detalha a adaptação do TOM Generator ao caso de estudo proposto pelo IRIT.
5. **TOM Editor** - Apresenta as fases de implementação e desenvolvimento do TOM Editor.
6. **Aplicação da TOM Framework** - Contém os detalhes de aplicação da *framework* a um caso de estudo.
7. **Conclusões e trabalho futuro** - Contém uma análise do trabalho desenvolvido e sugestões de trabalho futuro.

2

INTERFACES DO UTILIZADOR

As interfaces do utilizador (UI) fazem a ligação entre o utilizador e o computador, ou outro dispositivo tecnológico. Hoje em dia, quando estamos a olhar para o ecrã e a interagir com o sistema do computador temos acesso a várias funcionalidades que evoluíram ao longo do tempo.

É possível interagir com uma UI de várias formas, quer pela introdução de *inputs* ou pela interpretação dos *outputs* obtidos. A audição e a visão são dois exemplos de como os *inputs* e *outputs* podem ser sentidos na UI. Quando estamos a interagir com uma aplicação interativa, tipicamente através do ecrã, podemos ver os *outputs* gerados à medida que vamos enviando *inputs*, através do teclado ou rato, por exemplo [(Paiva, 2006)]. Por outro lado, o *output* não está limitado ao ecrã, sendo comum a utilização de outras modalidades tais como sons (em situações de erro, por exemplo)..

Com o emergir de novas interfaces do utilizador, a interação do utilizador com estas também foi evoluindo. Nas próximas seções vamos perceber como se realiza a interação do utilizador com os diferentes tipos de interfaces, como é que as mesmas foram evoluindo com o tempo e como é que se avaliam as interfaces gráficas de forma a garantir que o sistema está preparado para o utilizador.

2.1 INTERAÇÃO COM A UI DO SISTEMA

A interação dos humanos com as máquinas vai muito além do uso do rato e do teclado, existindo todo um processo interno por parte do utilizador para a execução das ações. Norman (1988) refere que existem sete etapas genéricas na interação habitual dos utilizadores com a interface de um sistema:

1. Definir o objetivo principal da interação.
2. Planear a forma de atingir o objetivo.
3. Especificar a sequência de ações para cumprir com o definido.

4. Executar a sequência de ações.
5. Observar o resultado da execução das ações (estado do sistema).
6. Interpretar o estado do sistema.
7. Verificar se o resultado da interpretação está de acordo com o objetivo inicialmente definido.

As etapas definidas anteriormente podem ser divididas em duas partes. Na primeira parte (etapas 1-4), é onde se executam as ações necessárias para atingir o objetivo definido, de seguida (etapas 5-7), avalia-se o resultado das ações no sistema. A Figura 2 demonstra esquematicamente o ciclo de execução de ações no sistema pelo utilizador.

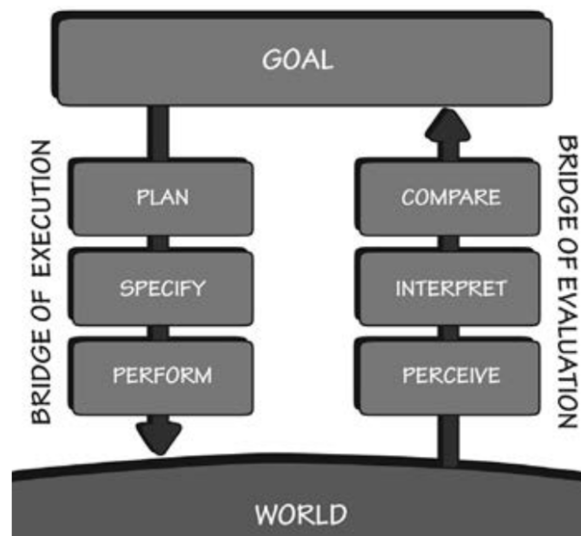


Figura 1.: As etapas do ciclo de interação (Norman, 1988).

2.2 TIPOS DE INTERFACES DO UTILIZADOR

O ciclo de interacção acima aplica-se a qualquer um dos tipos de interface actualmente mais comuns.

2.2.1 *Interface da linha de comandos*

As interfaces da linha de comandos (do inglês, *Command Line Interface (CLI)*) são uma das mais antigas formas de interação do utilizador com as máquinas [(Raymond, 2003)]. Permitem ao utilizador comunicar textualmente com o sistema do computador através da introdução de comandos a partir do teclado, obtendo-se de seguida uma resposta no ecrã do computador. Este sistema de introdução de comandos e obtenção de respostas, pode ser

definido como uma conversa entre o utilizador e a máquina.

Este tipo de interface tem ainda um grande peso na indústria, por exemplo, na gestão de infraestruturas e servidores de sistemas UNIX, e por utilizadores que pretendem ter acesso mais rápido, com recurso a atalhos, no manuseamento do sistema.

2.2.2 *Interface gráfica do utilizador*

As GUIs trouxeram novas possibilidades de interação dos humanos com as máquinas. Passou a ser possível utilizar imagens e gráficos como *input* ou *output* na comunicação com o sistema do computador. As primeiras GUIs permitiam dividir o ecrã em janelas, utilizar ícones para representar ficheiros e aplicações e usar menus para organizar o acesso aos comandos disponíveis. Para além disso, foi introduzido um novo dispositivo - rato, ainda usado nos dias de hoje, que permite a manipulação direta das janelas dos programas, a seleção de texto, cliques em elementos, etc... A manipulação da interface através do teclado também foi evoluindo com o tempo, com a introdução de atalhos no teclado e teclas especiais.

Este género de interface permite uma utilização mais simples e direta para todo o tipo de utilizadores, com um tempo de aprendizagem inferior ao necessário para trabalhar com uma CLI. No entanto, para os utilizadores mais exigentes e experientes, a velocidade de interação com o sistema pode diminuir, uma vez que é necessário executar mais etapas para conseguir realizar o objetivo. Este objetivo poderia eventualmente ser realizado numa CLI com recurso apenas a um comando.

2.3 EVOLUÇÃO DAS INTERFACES GRÁFICAS DO UTILIZADOR

Hoje em dia é quase impossível não ter um contacto diário com uma GUI, seja a partir do *smartphone*, do computador ou até mesmo nos carros mais recentes. As origens das GUIs remontam a 1968, quando Douglas Engelbart, inventor do rato, realizou uma demonstração ao público onde apresentou as capacidades de manipulação de uma interface gráfica (edição de documentos, suporte para várias janelas, etc...) [(Reimer, 2005)].

A Xerox foi a empresa que inicialmente desenvolveu o primeiro protótipo, o Xerox Alto, que juntava todos os elementos modernos das interfaces gráficas, como um rato de três botões, um ecrã *bit-mapped* e janelas gráficas. Mais tarde, a suceder esta versão surgiu o Star, uma versão comercial, com mais algumas inovações, como o duplo clique em ícones, sobreposição de janelas e caixas de diálogo [(Raymond and Landley, 2004)].

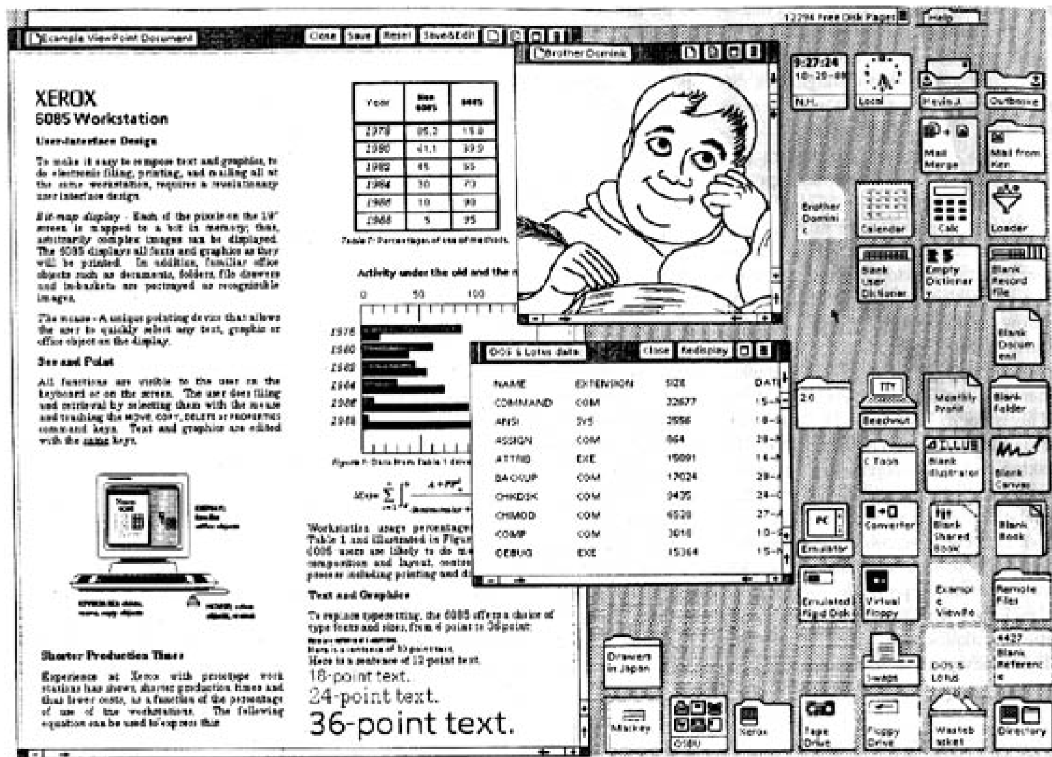


Figura 2.: *Screenshot* da interface Star (Raymond and Landley, 2004).

Subsequentemente, e uns alguns anos depois, a Apple e Microsoft lançaram no mercado interfaces gráficas próprias com os seus novos computadores. Estas interfaces, com o decorrer dos anos, ficaram mais intuitivas e mais fáceis de usar por parte do utilizador comum.

Neste momento, para além dos computadores, existem muitos dispositivos com interfaces gráficas, desde os telemóveis, *tablets*, *wearables*, às máquinas automáticas de tirar café e bebidas. Com toda esta expansão, tornaram-se necessárias normas e ferramentas que permitam avaliar se o comportamento da interface está de acordo com o previsto, para que o utilizador final não saia prejudicado com a interação com o dispositivo.

A evolução tecnológica que acompanhou o desenvolvimento das novas interfaces gráficas permitiu que as aplicações nativas (a executar directamente no sistema operativo do computador do utilizador) cedessem o lugar a aplicações baseadas em tecnologia web (*Hypertext Markup Language (HTML)* + *Cascading Style Sheet (CSS)* + *JavaScript (JS)*), disponíveis através de qualquer *browser*. Este género de aplicações não precisa de ser instalado, não exige muitas configurações, pode ser acedido em qualquer lugar e algumas aplicações web suportam e funcionam em modo offline, quando não existe ligação à internet. Por outro lado, as aplicações ditas nativas (as que correm directamente na máquina) são mais dependente da sistema operativo, sendo mais complicado adaptá-las a diferentes dispositivos e ambientes

de execução.

Atualmente, já é possível construir aplicações nativas com recurso a tecnologias web. A expansão que estas tecnologias tiveram nos últimos anos permitiu que surgissem ferramentas, como a *framework* Electron[1], por exemplo, que possibilita a construção de aplicações nativas para os diferentes sistemas operativos com base em tecnologia web.

As aplicações web são hoje em dia um importante meio de comunicação, de trabalho e lazer para a grande maioria de empresas e pessoas. Torna-se, de facto, importante garantir que as interfaces gráficas destas aplicações estejam construídas a pensar em todos os tipos de utilizadores, de forma a garantir que toda a interação ocorra de forma ágil e sem problemas.

2.4 AVALIAÇÃO DA USABILIDADE

Segundo Offutt (2002), os três principais fatores para o sucesso das aplicações na web são a usabilidade, a segurança e a fiabilidade. A usabilidade é um dos elementos de grande relevância na avaliação da qualidade das interfaces gráficas, assim como a acessibilidade, por assegurarem o controlo da interação do utilizador com a aplicação. Para Cruz and Campos (2013), as técnicas utilizadas na avaliação da usabilidade servem para identificar problemas que os utilizadores possam vir a sentir ao utilizarem as interfaces.

A usabilidade define a qualidade de utilização de uma interface gráfica. Se a usabilidade for tida em conta durante o período de desenvolvimento de uma interface, é possível eliminar alguns problemas, como por exemplo, disponibilizar facilmente na aplicação todas as informações esperadas pelo utilizador [(Winckler and Pimenta, 2002)].

Existem dois tipos de métodos utilizados na avaliação da usabilidade. Os métodos empíricos, são caracterizados pela necessidade de participação dos utilizadores no decorrer da avaliação, sobre a observação de um perito. Já os métodos análiticos são conhecidos por serem realizados por especialistas com conhecimentos sobre a qualidade das interfaces gráficas.

2.4.1 *Métodos empíricos*

Estes métodos baseiam-se principalmente na observação dos utilizadores finais. Aquando da realização da avaliação, esta pode ser feita com base num protótipo ou com recurso ao

1 http://electron.atom.io/ (visitado pela última vez em 04/10/2016).

sistema final. Os protótipos são normalmente utilizados no decorrer do desenvolvimento do sistema, e tem por objetivo identificar problemas de utilização de forma a que sejam corrigidos antes de sair uma versão final do sistema, sendo esta etapa designada por avaliação formativa. Quando uma versão final do sistema está disponível para ser utilizada por utilizadores entramos na etapa da avaliação sumativa, que tem como objetivo verificar se a qualidade do sistema está de acordo com um determinado padrão.

Apesar dos benefícios referidos anteriormente, o custo de aplicação destes métodos é normalmente muito elevado. A necessidade de recrutar pessoas adequadas, conseguir obter condições de utilização realistas, recolher e, no final, analisar os dados da utilização do sistema fazem com que o custo e o tempo sejam, especialmente, elevados.

É importante sabermos o que medir na avaliação da usabilidade. Moore and Redmond-Pyle (1995), sugere uma forma comum de medição da usabilidade:

TESTES DE DESEMPENHO

Enquanto os utilizadores interagem com o sistema na execução de uma tarefa, utilizam a eficácia como medida. As métricas usuais são a precisão, velocidade e os erros.

SONDAGEM - *"ATTITUDE SURVEY"*

Avalia-se a satisfação e a percepção do utilizador, com recurso a questionários ou entrevistas.

Algumas das técnicas empíricas de avaliação da usabilidade são apresentadas a seguir:

THINKING ALOUD

Em 1993, Ericsson and Simon (1993) descreveu esta técnica. Mais tarde, a mesma foi desenvolvida por van Someren et al. (1994). Este processo consiste em estimular os utilizadores a expressar, no decorrer do processo de avaliação do sistema, todos os seus pensamentos. Desta forma, é possível capturar e analisar mais tarde, não só todos os eventos de utilização, como o processo cognitivo por trás deles: dúvidas, dificuldades, raciocínios, expectativas, o que os utilizadores sentiram, etc.

QUESTIONÁRIOS

Segundo Helander et al. (1997), depois de o utilizador ter contacto com o sistema, é possível serem-lhe colocadas algumas questões sobre a sua experiência de utilização. Os questionários podem variar de formalidade, isto é, podem ter questões mais genéricas e espaços para comentários, até questões mais específicas sobre aspectos que se pretendem avaliar. Os questionários têm um papel bastante útil na avaliação

das interfaces e podem ser constituídos por diferentes formatos, com diferentes escalas, como sugerido em Dix et al. (2004). Fornecem um método fácil na obtenção de dados sobre a utilização do sistema.

ENTREVISTAS

As entrevistas permitem obter feedback sobre o sistema, de forma simples. Dependendo dos objetivos definidos, a estrutura da entrevista pode variar, entre entrevista livre, semi-estruturada ou estruturada.

2.4.2 *Métodos analíticos*

Os métodos analíticos ou de inspeção, ao contrário dos métodos de empíricos, não necessitam de utilizadores reais, tem um custo de aplicação mais baixo, e são realizados por vários especialistas de usabilidade. O objetivo principal é prever possíveis problemas de usabilidade num sistema, sendo bastante úteis na validação de decisões na fase inicial do desenvolvimento.

Existem várias técnicas utilizadas neste método, como a avaliação heurística, *cognitive walkthrough* e *software guidelines* (Paterno, 2000). São normalmente utilizadas de forma manual por especialistas externos à equipa de desenvolvimento.

AVALIAÇÃO HEURÍSTICA

Esta técnica, utiliza uma lista específica de heurísticas para a detecção de possíveis problemas na interface. Na próxima lista são apresentados as dez heurísticas da usabilidade, desenvolvidas por Nielsen and Molich (1990):

1. **VISIBILIDADE DO ESTADO DO SISTEMA**
   O sistema deverá manter sempre o utilizador informado sobre o que está a acontecer, através do *feedback* fornecido.
2. **COMPATIBILIDADE DO SISTEMA COM O MUNDO REAL**
   O sistema deve falar a linguagem do utilizador, com palavras, frases e conceitos familiares, em vez de termos mais informáticos. Seguir as convenções do mundo real, com a informação a aparecer com uma ordem natural e lógica.
3. **CONTROLO DO UTILIZADOR E LIBERDADE**
   Os utilizadores, por vezes, enganam-se ao escolher determinada função do sistema, e nestas situações é necessário ter as saídas claramente identificadas para voltar ao estado anterior.
4. **CONSISTÊNCIA E NORMAS**
   Os utilizadores não devem ter de adivinhar se ações diferentes, significam a mesma coisa. Devem ser seguidas as normas do sistema.

5. **PREVENÇÃO DE ERROS**
   O sistema deve prevenir, sempre que possível, a ocorrência de erros. As condições mais propensas ao erro não devem ser apresentadas ao utilizador.
6. **RECONHECIMENTO EM VEZ DE MEMORIZAÇÃO**
   O utilizador não deve ter que se lembrar da informação de uma parte da interface do sistema para outra. As instruções para utilização do sistema devem estar visíveis.
7. **FLEXIBILIDADE E EFICIÊNCIA DE UTILIZAÇÃO**
   Permitir que utilizadores mais experientes utilizem atalhos para acelerar a interação com o sistema e acesso direto para as funcionalidades mais utilizadas.
8. **ESTÉTICA E DESIGN MINIMALISTA**
   As informações mostradas ao utilizador não devem ser irrelevantes ou desnecessárias. A informação deve ser apresentada de forma simples e intuitiva.
9. **AJUDAR OS UTILIZADORES A RECONHECER, DIAGNOSTICAR E A RECUPERAR DE ERROS**
   As mensagens de erro visíveis ao utilizador devem ser expressas em linguagem simples, indicar qual o tipo de erro encontrado e sugerir uma possível solução.
10. **AJUDA E DOCUMENTAÇÃO**
   Mesmo que seja melhor para o sistema não ser usado com documentação, em algum momento pode ser necessário recorrer à mesma, e é bom que a mesma esteja disponível.

Para se realizar a análise heurística às interfaces gráficas é necessário seleccionar um pequeno conjunto de especialistas, normalmente entre 3 a 5. Como estão vários especialistas a examinar a interface é mais provável encontrar problemas diferentes, que são consolidados no relatório elaborado. A avaliação realiza-se desde as fases iniciais do projeto, com protótipos, onde se compara a interface com a lista de heurísticas anteriores.

COGNITIVE WALKTHROUGH

É uma técnica focada na forma em como os utilizadores exploram a interface para aprender a utilizar as funcionalidades fornecidas. Baseada na teoria de aprendizagem exploratória de Polson et al. (1992), é utilizada por peritos a partir da fase inicial de prototipagem da interface, e encontra-se estruturada em 4 questões:

1. A ação é suficientemente evidente para o utilizador?
2. O controlo para executar a ação encontra-se visível?
3. O utilizador consegue associar a ação correta ao controlo?

4. É possível ao utilizador interpretar de forma correta a resposta do sistema à ação escolhida? O *feedback* é adequado?

Antes de se começar aplicar esta técnica é necessário definir as ações necessárias para se cumprir uma tarefa. Posteriormente, é necessário responder, antes e depois da execução da ação, às questões referidas anteriormente. Esta técnica pode ser utilizada com vários especialistas.

2.5 Avaliação da qualidade de implementação

As técnicas da secção anterior focam-se na usabilidade, procuram garantir a qualidade da concepção da interface. No entanto, são sensíveis a problemas de fiabilidade. Por um lado, problemas de qualidade na implementação vão alterar para pior a usabilidade do sistema inicialmente prevista nos testes formativos (efectuados em protótipos). Por outro, problemas de fiabilidade durante a aplicação dos testes sumativos interferem com a avaliação da qualidade da concepção.

As técnicas focadas no teste do sistema preocupam-se com o comportamento geral da aplicação. Existem dois tipos de abordagens que se podem seguir para a realização dos testes: *white box* e *black box*. Nos testes *white box* é possível analisar a lógica interna do sistema, mas é necessário ter conhecimentos sobre o código desenvolvido. Os testes *black box* tem como intuito serem completamente independentes do comportamento interno e à estrutura do sistema. Como tal, preocupa-se em encontrar situações em que o comportamento esperado não está de acordo com as especificações (Myers, 2004).

Em Paiva (2006), são referidas várias abordagens para a realização de teste de software. Os testes baseados em modelos são uma técnica *black box* que comparam o comportamento e o estado de um software com o seu modelo, permitindo assim a detecção de desvios ao modelo. Esta técnica assume um relevo importante no contexto desta dissertação, porque permite avaliar a qualidade de implementação do software (Cruz and Campos, 2013).

2.6 Testes baseados em modelos para interfaces gráficas

Como referido na Seção 2.4, as principais técnicas utilizadas na realização de avaliações a interfaces gráficas baseiam-se na observação do comportamento e nas ações dos utilizadores enquanto interagem com a interface gráfica do sistema. Quando se coloca uma GUI sobre avaliação pretende-se obter uma validação, isto é, perceber se a mesma está a cumprir os objetivos especificados e, eventualmente, encontrar falhas.

Nos testes baseados em modelos cria-se um modelo abstrato do comportamento esperado do utilizador do sistema sob teste (do inglês, *System Under Test (SUT)*), de forma a simular a interação numa GUI. Através do modelo do SUT é possível gerar, automaticamente e com o mínimo esforço, um grande número de casos de teste. Este é ainda utilizado como o oráculo para verificar se a implementação sob avaliação passou no teste (Jonathan Jacky et al., 2008).

2.6.1 *Processo de MBT*

A geração de casos de teste com MBT pode ser realizada de duas formas distintas, *offline* ou *online*. Na abordagem *offline* desta técnica os casos de teste são gerado antes de serem executados. O modo *online* permite que os casos de teste sejam gerados enquanto os testes são executados.

A utilização de uma ferramenta na geração de casos de teste é influenciada pelos critérios de cobertura definidos pelo especialista. A alteração dos critérios origina a geração de casos de testes distintos. O processo, representado na Figura 3, encontra-se dividido em cinco etapas, de acordo com Utting and Legeard (2007):

1. Modelação do SUT
2. Gerar testes abstratos a partir do modelo
3. Transformar os testes abstratos em testes executáveis
4. Executar os testes
5. Analisar os resultados obtidos do teste

Na primeira etapa do processo o modelo abstrato do SUT deve ser criado, este modelo deve ser mais simples que o SUT e estar de acordo com as especificações do sistema.

Após a modelação terminar, são gerados os testes abstratos a partir do modelo, com base nos critérios definidos. A terceira etapa consiste em transformar os testes abstratos em testes executáveis. Os testes gerados estão expressos em termos do modelo e é necessário transformá-los em testes executáveis (concretos), construindo-se uma ligação entre os testes e o sistema. Na quarta etapa os testes são executados no software. Na etapa final é realizada a análise dos resultados, verificando-se a validade dos mesmos.

Nos testes *online* as etapas 2, 3 e 4 estão geralmente juntas num único passo. Na quinta e última etapa os resultados dos testes são analisados, comparando-se estes resultados com o modelo, para perceber se está de acordo com os resultados esperados ou existe algum problema (Utting and Legeard, 2007).

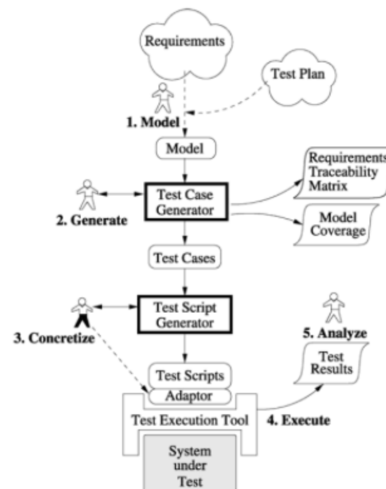


Figura 3.: Processo de *Model Based Testing* (Utting and Legeard, 2007).

2.6.2 *Benefícios e dificuldades*

Ao longo dos últimos anos, vários estudos (Dhawan et al., 2011; Mikko Aleksi Makinen, 2007) tem comprovado que o MBT é bastante eficaz, principalmente quando é utilizado para testar aplicações pequenas, sistemas integrados e interfaces gráficas. Alguns dos benefícios que se destacam na utilização de MBT são os seguintes:

- O modelo do sistema pode ser construído mais cedo, com base nas especificações do sistema.
- Geração de uma enorme quantidade de casos de teste, a um baixo custo.
- As alterações dos critérios possibilitam a geração de diferentes casos de teste.
- Processo bastante automatizado, que permite reduzir o tempo dos testes.
- As alterações ao modelo do sistema são facilmente adaptadas e testadas. Os modelos podem ainda ser reutilizáveis no futuro.
- A detecção de possíveis problemas acontece mais cedo, sendo o código corrigido mais rápido.

Apesar dos claros benefícios enunciados anteriormente, no MBT existem algumas dificuldades ou problemas, tais como:

- A necessidade de uma especificação formal do modelo para se conseguir realizar a geração dos testes.

- As alterações no modelo do sistema podem resultar em casos de teste completamente diferentes do expectável.
- A modelação do sistema é um processo trabalhoso e complicado.
- Problemas com a explosão de estados, tornando a manutenção do modelo díficil e complicando a geração dos testes.

2.6.3 *Trabalhos relacionados*

No decorrer desta dissertação foi realizada uma pesquisa de ferramentas que seguissem uma abordagem similar ao processo de MBT a interfaces gráficas do utilizador, tanto para aplicações nativas como aplicações web.

O GUITAR (Nguyen et al., 2014), é uma ferramenta que suporta uma extensa variedade de técnicas de teste a interfaces gráficas de múltiplas plataformas (*JFC, SWT, Web, Android*). Construído com base numa arquitetura modular, possibilita a utilização de *plugins*, tornando-se mais extensível e flexível. O modelo do sistema é gerado com base na informação extraída automaticamente da GUI, sendo posteriormente utilizado para gerar casos de teste. Posteriormente é realizada a execução dos testes. Uma das fraquezas encontradas nesta ferramenta é o facto de não ser possível alterar manualmente o modelo obtido.

O Spec Explorer[2] é uma ferramenta de MBT de geração e execução de casos de teste a aplicações nativas. Paiva (2006) explorou a aplicação de Spec Explorer ao teste de interfaces gráficas. Uma das dificuldades que identificou foi a construção manual do código de mapeamento com as GUIs (Paiva et al., 2005). Para resolver este problema foi construída a GUI Mapping Tool, integrada com o Spec Explorer, auxiliando o utilizador a relacionar as ações lógicas do modelo com ações físicas na GUI, gerando automaticamente no final a informação sobre o mapeamento.

Outra alternativa, proposta por Moreira et al. (2013), é o *Pattern Based GUI Testing (PBGT)* que utiliza o PARADIGM, uma linguagem de domínio especifício (do inglês, *Domain Specification Language (DSL)*), na modelação do sistema. Esta abordagem baseia-se na identificação de padrões na interface, existindo para cada padrão um conjuntos de testes pré-definidos.

MISTA (Xu, 2011), é uma ferramenta de geração de casos de teste que suporta várias linguagens (C++, C#, HTML, VB), não fazendo automaticamente a execução dos mesmos. Usa uma notação visual que permite criar modelos. Recorre a redes de Petri para

2 https://msdn.microsoft.com/en-us/library/ee620411.aspx (visitado pela última vez em 12/10/2016).

a representação dos modelos, onde é possível especificar os dados e controlos dos objetos.

Em Silva et al. (2008) é proposta uma nova abordagem com base em modelos de tarefas. Com recurso a estes modelos e através do recurso ao TERESA (Berti et al., 2004) é gerada automaticamente uma máquina de estados. A partir deste é criado um grafo, onde se obtém os casos de teste, em Spec#. Esta abordagem foi utilizada para avaliar aplicações nativas.

2.6.4 *Tecnologias relacionadas*

Como existe a necessidade de se facilitar a tarefa de criação dos modelos do sistema, foi realizada uma pesquisa sobre *plugins* do *browser* que permitissem de alguma forma construir modelos ou cenários de teste. Foram encontradas algumas ferramentas de captura da interação do utilizador, conhecidas como *capture-reply*, onde é possível, em algumas, introduzir validações no cenário. As principais extensões que se destacaram foram:

- SeleniumHQ[3], um conjunto de ferramentas de software para apoiar a automação de testes a aplicações web. O Selenium IDE é uma das ferramentas mais conhecidas do conjunto, que a partir de uma extensão no browser Firefox consegue criar casos de teste em Selenium ou em outro tipo de *output* pretendido (C#, Java, Perl, PHP, Ruby, Python e Perl).
- TESTIM[4], é uma plataforma que permite a automação dos testes, através de uma extensão no Google Chrome. Tem como principal funcionalidade a captura da interação do utilizador com a aplicação web, construindo um cenário de teste automaticamente. Este cenário, pode ser modificado com novas validações e, mais tarde, executado no browser.
- O Chromium Browser Automation[5] é uma extensão para o Google Chrome que tem como objetivo facilitar ao utilizador as tarefas diárias de utilização de algumas páginas web. Através da captura da interação do utilizador com uma página web, cria um cenários de evento que posteriormente o utilizador pode executar.

2.7 CONCLUSÕES

Neste capítulo foram abordados alguns aspectos das interfaces do utilizador. A sua evolução, a forma como o utilizador habitualmente interage com as mesmas, ligação homem máquina, os tipos de interface e os métodos que se utilizam para avaliar as interfaces gráficas, foram

3 http://www.seleniumhq.org/ (visitado pela última vez em 14/10/2016).
4 https://www.testim.io/ (visitado pela última vez em 14/10/2016).
5 http://chrome-automation.com/ (visitado pela última vez em 14/10/2016).

alguns dos aspectos importantes descritos.

A avaliação das interfaces gráficas é um componente bastante importante no processo de desenvolvimento de software porque permite perceber se a mesma está de acordo com as especificações, se tem uma apresentação agradável e é funcional para os utilizadores finais. A avaliação, dependendo dos métodos escolhidos, tende a ser dispendiosa em termos de custo e tempo.

A usabilidade é um aspecto bastante útil para se avaliar a qualidade de uma interface gráfica, mas que pode ser afetada, quando se usa como base da avaliação uma versão do projeto desenvolvido com erros de implementação (sem fiabilidade). Desta forma, é importante que existam técnicas que consigam inferir sobre a qualidade do código implementado no sistema, através da utilização da interface gráfica. Esta avaliação não avalia a usabilidade, mas garante que quando for realizada a avaliação da usabilidade, as inconsistências possíveis do sistema não vão acontecer.

Os testes baseados em modelos são uma técnica que permite automatizar o processo de geração e execução de casos de teste a interfaces gráficas do utilizador, com um custo inferior às técnicas manuais. Permite encontrar inconsistências no sistema, isto é, desvios relativamente ao modelo especificado. Esta técnica apresenta um problema de custo de construção do modelo de suporte à geração dos casos de teste, sendo que a grande maioria das ferramentas não suporta um modelo gráfico que facilite a construção do mesmo.

Foram ainda apresentadas algumas ferramentas de MBT e outras de *capture-reply*. Estas últimas serviram para avaliar o modo de como se pode vir a construir o modelo do sistema de forma mais automática, com recurso a um novo *plugin*. Desta forma, o custo de construção irá diminuir, tornando a técnica de MBT mais fluída e rápida.

3

TOM FRAMEWORK

A TOM Framework é constituída por um conjunto de ferramentas flexíveis que pretendem facilitar a avaliação da qualidade de aplicações web no geral, mas que pode eventualmente ser aplicado a outros ambientes, como veremos no Capítulo 4. Encontra-se dividida em dois componentes principais: o TOM Generator, responsável pela geração e execução de casos de teste, e o TOM Editor, responsável pela criação e manutenção dos modelos e do seu mapeamento com as aplicações sob teste. O primeiro foi desenvolvido por Rodrigues (2015) e é descrito neste capítulo. A concepção e desenvolvimento do segundo são descritas no Capítulo 5.

O TOM Generator é uma ferramenta modular, que faz a leitura e interpretação dos ficheiros que modelam o sistema, gerando posteriormente casos de teste sobre a interface gráfica sob teste. No final, é possível executar os casos de teste na aplicação web. Esta ferramenta foi desenhada a pensar num baixo custo de manutenção e em futuras extensões de desenvolvimento, e como tal encontra-se organizada de forma modular, deste modo a substituição ou alteração de um módulo têm um impacto limitado nos outros módulos do sistema (Rodrigues, 2015). Esta ferramenta está preparada para realizar a geração e execução de casos de teste a interfaces gráficas de aplicações web, mas se existir uma adaptação dos módulos será possível avaliar aplicações nativas.

Internamente, o processo de comunicação entre os módulos encontra-se dividido em várias etapas, seguindo uma abordagem de MBT, explicada na Seção 2.6.1. O processo do TOM Generator (ver Figura 4) encontra-se dividido em 6 etapas:

1. Leitura e interpretação do modelo do sistema.
2. Geração de um grafo a partir do modelo do sistema.
3. Aplicação de algoritmos de travessia sobre o grafo obtido anteriormente, gerando-se sequências de testes abstratos normais e, se solicitado, com mutações.
4. Geração de casos de teste executáveis.
5. Execução dos testes na interface gráfica.

6. Análise dos resultados obtidos.

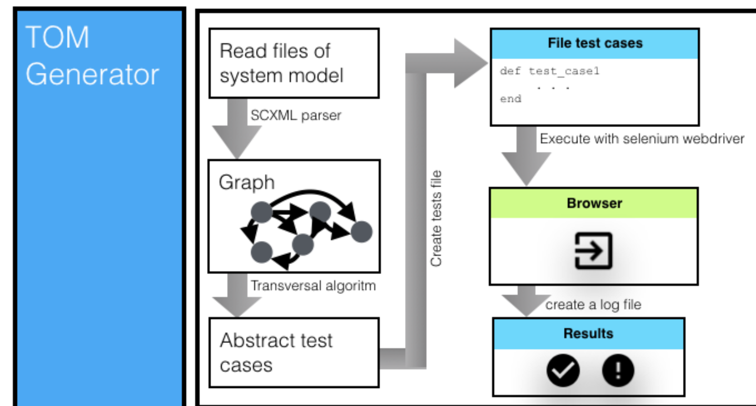


Figura 4.: Processo de MBT do TOM Generator

Nas próximas seções pretende-se explorar detalhadamente o processo definido na Figura 4, com ênfase nos detalhes técnicos das validações e mutações.

3.1 MODELAÇÃO DO SISTEMA

O modelo do sistema é uma parte crucial no processo de testes baseados em modelos. No caso da interfaces gráficas o modelo exprime os aspectos chaves da interação do utilizador com estas (Paterno, 2000).

O sistema é definido com recurso a máquinas de estado, que, neste caso, representam a interação e navegação do utilizador na interface gráfica de uma aplicação web. Assim, cada página web é representada através de um estado. Em cada estado podem existir transições para novos estados, correspondentes a cliques em links para outras páginas, formulários onde o utilizador preenche com dados as informações solicitadas na página e validações quando se pretende validar a existência de certos elementos na página web.

3.1.1 *Elementos principais da máquina de estados*

As máquinas de estados são modelos utilizados recorrentemente na engenharia de software. São vistas como uma forma útil de pensar sobre o comportamento dos sistemas, desde a fase inicial do desenho até ao teste de software (Belli, 2001).

Para modelar o comportamento da GUI vamos recorrer a *State Chart XML (SCXML)*, uma linguagem que fornece elementos para descrever máquinas de estados baseados em eventos,

de forma limpa e organizada. Os elementos principais de SCXML que são utilizados para descrever a interação e validação na interface gráfica são os seguintes:

<STATE>
É o elemento principal do modelo e representa um estado. Os próximos elementos, a apresentar, existem dentro deste elemento. Representa habitualmente uma página web. Quando se está a preencher um formulário na interface gráfica utiliza-se este elemento dentro do atual estado, criando-se assim um sub-estado.

<TRANSITION>
É utilizado quando estamos perante um evento de transição entre estados. Este evento pode ter origem em cliques de *links*, botões ou submissão de formulários. Dentro deste elemento podem existir os seguintes elementos, *step*, *submit* e *error*, sendo explicada a seguir a necessidade de utilização de cada um.

<STEP>
É utilizado quando é necessário mais do que uma ação para atingir um determinado estado. Por exemplo, quando é necessário abrir a barra de navegação e clicar em um *link* que nos leva para uma nova página web. O *step* nesta situação acontece porque é necessário primeiramente haver um clique para abrir o menu.

<SUBMIT>
Quando se está a preencher um formulário síncrono é necessário no final fazer a submissão ao servidor que nos leve a uma nova página. Este elemento define qual o estado para o qual se vai transitar depois da submissão.

<ERROR>
A *framework* tem a particularidade de criar e aceitar mutações no preenchimento de formulários, tal como explicado mais à frente. Caso a mutação escolhida interfira com o decorrer normal do teste é necessário definir neste elemento o estado para o qual se vai transitar.

<SEND>
Este elemento é utilizado para representar a introdução de informação do utilizador no preenchimento dos formulários (*inputs*).

<ONENTRY>
Utilizado para representar uma validação na entrada de uma página web, isto é, na entrada de um estado.

<ONEXIT>
É similar ao elemento anterior, mas a validação ocorre quando se está a sair de um determinado estado.

Estruturalmente, os elementos descritos anteriormente podem ser combinados tal como ilustrado no exemplo 3.1, onde é percetível a estrutura hierárquica da máquina de estados. Nesta estrutura ainda não estão representados os atributos de cada elemento.

```
<state ... >
  <state ... >
    <send ... />

    <transition ... >
      <submit ... />
      <error ... />
    </transition>

    <onexit ... />
  </state>

  <transition ... />
  <onentry ... />
  <onexit ... />
</state>
```

Listing 3.1: Exemplo da estrutura dos elementos.

3.1.2 *Atributos dos elementos da máquina de estados*

Como anteriormente referido, por si só os elementos da máquina de estados não conseguem transmitir informação sobre o tipo de ações que o utilizador executa ou deve executar na interface gráfica. Como tal, é necessário recorrer a atributos para definir essa informação. Nas tabelas que se seguem iremos encontrar os atributos que fazem parte de cada elemento, e quando e como devem ser utilizados. A Tabela 1, apresenta os atributos do elemento *state*, as Tabelas 2 e 3, os atributos do elemento *transition*, a Tabela 4, os atributos do elemento *send*, e, finalmente, na Tabela 5, são apresentados os atributos dos elementos *onentry* e *onexit*.

<state id="" type=""> ... </state>	
id	Atributo obrigatório que é utilizado para identificar cada estado.
type	Atributo utilizado quando se está a preencher um formulário na interface gráfica, tomando o valor *"form"*. Caso contrário, não é definido.

Tabela 1.: Tabela com os atributos do elemento *state*.

<transition id="" label="" />	Para transições simples entre páginas.
id	Atributo obrigatório que se utiliza como identificador da ação.
target	O *id* do estado para o qual se pretende transitar.

Tabela 2.: Tabela com os atributos do elemento *transition*.

<transition type="" label="" > <submit target=""/> <error target="" /> </transition>	Género de transição que é utilizado quando se está a modelar o preenchimento de um formulário na interface gráfica. O elemento error é opcional, e usa-se em caso de mutação de falha.
type	Serve para identificar o tipo de transição da página web, depois da submissão do formulário. Assume o valor de *"form"* quando se preenche um formulário normal, *"alert"* quando surge uma *alert window* e *"ajax"* para eventos assíncronos.
label	Atributo utilizado para representar identificar a ação de submissão de um formulário.
target	O *id* do estado para o qual se pretende transitar.

Tabela 3.: Tabela com os atributos do elemento *transition* nos formulários.

<send label="" type="" element="" />	Correspondem à introdução de informação nos formulários.
label	É utilizado como identificador da ação.
type	Indica se a informação a ser introduzida no formulário é obrigatória ou opcional. *"required'* ou *"optional"* são os valores possíveis.
element	Este atributo é somente utilizado quando a informação que se vai adicionar ao formulário é do tipo *checkbox* ou *selectbox*, sendo estes os valores possíveis de entrada.

Tabela 4.: Tabela com os atributos do elemento *send*.

<onentry id="" type=""/> <onexit id="" type=""/>	Corresponde às validações que são realizadas quando se entra ou sai de uma página web.
id	Utilizado como identificador da ação.
type	O tipo de validação que pode ser efetuada. Os valores que este atributo aceita são descritas na Seção 3.3.

Tabela 5.: Tabela com os atributos dos elementos *onentry* e *onexit*.

Na Listagem 3.2, encontramos um exemplo de modelação dos atributos dos elementos no preenchimento de um formulário de registo de um utilizador. Quando a página de registo é apresentada são enviados **(send)** para os campos do formulário os dados referentes ao *email*, *password* e confirmação de *password*. Quando a informação é submetida com sucesso **(transition - submit)** deverá ser apresentada a página de login, caso exista algum erros nos dados dos campos deste formulário deverá ser apresentada uma página que contém os erros do registo **(transition - error)**. Neste exemplo ainda não foram introduzidas as anotações de validação (*onentry* e *onexit*) porque ainda não foram apresentadas os diferentes tipos de verificação existentes (ver Secção 3.3).

```
<state id="register" >
    <state id="register_form" type="form" >
        <send label="email" type="required" />
        <send label="password" type="required" />
        <send label="conf_password" type="required" />

        <transition  type="form" label="submit_register" >
            <submit target="login_page" />
            <error  target="register_error" />
        </transition>
    </state>
</state>
```

Listing 3.2: Exemplo de modelação dos atributos.

3.2 MAPEAMENTO E DADOS

Ao longo da máquina de estados existem atributos que apenas servem para identificar as ações. É então necessário criar um mecanismo que permita guardar a informação da ligação entre a máquina de estados, o elemento HTML e ainda a informação dos dados do formulário e validações. As informações necessitam de estar guardadas em ficheiros separados, seguindo determinadas regras de construção, para que sejam processadas no TOM Generator.

Há medida que se vai criando a máquina de estados, ou depois de esta estar terminada, é necessário criar uma forma de mapear o comportamento definido na máquina de estados com a aplicação web. Para tal, recorremos a um novo ficheiro que contém a informação de como aceder ao elemento HTML e o tipo de ação a executar neste.

Como se pode verificar na Tabela 6, o ficheiro de mapeamento contém, para cada variável de identificação do modelo, uma configuração para determinar qual o tipo de elemento HTML da *Document Object Mode (DOM)* a encontrar, como encontrá-lo, a ação e o tipo de execução. O ficheiro utiliza a notação *JavaScript Object Notation (JSON)* na construção dos mapeamentos.

"#id" : {
how_to_find: "",
what_to_find: "",
what_to_do: "",
type_of_action: ""
}
	O formato genérico utilizado na mapeamento entre a interface gráfica e o modelo do sistema.

how_to_find	O localizador que se deve utilizar para encontrar o elemento na interface gráfica. Pode tomar os seguintes valores: *id, xpath, cssSelector, className, linkText, name, tagName, partialLinkText*.
what_to_find	Valor do tipo de elemento a ser encontrado.
what_to_do	O tipo de ação que será executada neste mapeamento. É um atributo opcional e os próximo valores são os únicos válidos neste campo: *sendKeys, click, submit, moveToElemetn, getText, accept*.
type_of_action	Auxilia na determinação do tipo de elemento HTML. É um atributo opcional que pode ser utilizado com um dos seguintes valores: *textBox, selectBox, checkBox*, ou com um elemento HTML.

Tabela 6.: Tabela com a descrição dos elementos de mapeamento do modelo.

A Listagem 3.3 mostra-nos o mapeamento de alguns elementos da Listagem 3.2 com a página web, nomeadamente do campo de *email* para o envio de informação, do botão de submissão do formulário e da validação existente no campo de *email*, com a utilização dos atributos definidos na tabela anterior.

```
[
  "email" : {
      "how_to_find" : "cssSelector",
      "what_to_find" : "div.form-group␣>␣#user_email",
      "what_to_do" : "sendKeys",
      "type_of_action" : "textbox"
  },
  "password" : {
      "how_to_find" : "cssSelector",
      "what_to_find" : "div.form-group␣>␣#user_password",
      "what_to_do" : "sendKeys",
      "type_of_action" : "textbox"
  },
  "conf_password" : {
      "how_to_find" : "cssSelector",
      "what_to_find" : "div.form-group␣>␣#user_conf_password",
      "what_to_do" : "sendKeys",
      "type_of_action" : "textbox"
  },
  "submit_register" : {
      "how_to_find" : "xpath",
      "what_to_find" : "(//button[@type='submit'])[2]",
      "what_to_do" : "submit"
  },
  "message_new_register": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV",
    "what_to_do": "getText"
  },
  "section_title_signup": {
    "how_to_find": "className",
    "what_to_find": "section-title",
    "what_to_do": "getText"
  },
  "attr_email" : {
      "how_to_find" : "xpath",
      "what_to_find" : "//form/div[2]/input",
      "type_of_action" : "id"
  },...
]
```

Listing 3.3: Exemplo de mapeamentos dos elementos.

Alguns dos elementos da máquina de estado, **send**, **onentry** e **onexit**, necessitam do apoio de um ficheiro que permita guardar os valores que são enviados para os formulários, ou que são utilizados para validar o texto ou elementos da página. É então necessário criar mais um ficheiro que contenha os valores dos dados que vão ser utilizados para o envio de dados na geração de casos de teste. A Tabela 7 descreve-nos a estrutura seguida.

[{ ”#id” : ”#value” } ...]	
#id	Identifica a ação do utilizador no ficheiro de valores.
#value	O valor que será utilizado para preencher os formulários ou nas validações que usem texto para comparação.

Tabela 7.: Tabela descritiva da estrutura do ficheiro de valores.

A Listagem 3.4 mostra-nos um exemplo do ficheiro de valores, com base no modelo definido na listagem 3.2, onde para cada *id* da ação existe um valor que irá ser utilizado no envio de dados para um formulário ou na validação de algum elemento na página web. Neste exemplo, a ação denominada por ”email”, que corresponde ao envio da informação de *email* para o formulário, irá utilizar o valor ”user@mail.com”. A ação ”password” e ”conf_password” são muito similares à anterior, neste caso é utilizado o valor ”1234567890” para o campo *password* e *conf_password* do formulário. Já a ação ”attr_email”, que corresponde à verificação do valor de um atributo de um elemento HTML, contém o valor esperado desse atributo, neste caso ”user_email”. A ação ”section_title_signup” contém o valor que deve ser utilizado para verificar o nome da secção.

```
[
  {   "email" : "user@mail.com"                 },
  {   "password": "1234567890"                  },
  {   "conf_password": "1234567890"             },
  {   "section_title_signup": "Signup␣section"  },
  {   "attr_email" : "user_email"               },...
]
```

Listing 3.4: Exemplo do ficheiro de valores.

3.3 VALIDAÇÕES

Um dos componentes mais essenciais num caso de teste são as validações. Sem estas não era possível perceber se existem situações anômalas que afetem a qualidade do software.

No momento em que um caso de teste está a ser executado, as validações verificam se as propriedades de saída obtidas no teste respeitam o conjunto predefinido de dados esperados. Quando se encontra um resultado diferente do esperado estamos perante uma falha que precisa de ser verificada pelo utilizador, para perceber se a falha é resultado de algum problema com o software.

As validações são introduzidas por norma dentro de cada estado, no modelo do sistema, e são representadas pelos elementos SCXML *onentry* e *onexit*, que respectivamente fazem a validações ao entrar e sair de cada estado. O TOM Framework está equipado com as seguintes características de validação a uma interface gráfica:

DISPLAYED? / NOT_DISPLAYED?
Verifica se um dado elemento está visível/não está visível. Por exemplo, se o logo da aplicação está a aparecer na página.

IS_SELECTED / IS_NOT_SELECTED
Confirma se um elemento numa *selectbox* ou *checkbox* está selecionado/não está selecionado.

ENABLED? / DISABLED?
Testa se um dado elemento se encontra ativo ou desativo. Este teste pode ser utilizado quando temos botões que ficam ativos e desativos, conforme o tipo de interação do utilizador.

ATTRIBUTE
Verifica o valor do atributo de um elemento HTML, como por exemplo, o valor do atributo value de um *input*.

CSS
Verifica uma propriedade CSS de um elemento HTML. Por exemplo, o *backgroud-color*, *position*, etc...

CONTAINS
Testa se um elemento contém um determinado valor.

REGEX
Verifica se um elemento contém um valor que corresponda a uma expressão regular.

URL
Confirma o url de uma página.

DEFAULT
Verifica se um elemento tem exatamente um dado valor.

Na Listagem 3.2 foi apresentada um exemplo de modelação incompleto, visto que ainda não tinham sido introduzidas as validações essenciais para a avaliação da interface gráfica. A Listagem 3.5 que se segue, apresenta uma validação de entrada (*onentry*), quando a página de registo é apresentada, valida-se o título da seção, e duas validações de saída dos estados (*onexit*). A verificação efetuada no estado "register" verifica se um valor existe num atributo do elemento HTML. Na saída do estado "register_form" é verificado se uma

mensagem está a aparecer na página web. Pode-se consultar as Listagens 3.3 e 3.4 para ter o quadro completo dos dados necessários para o exemplo apresentado.

```
<state id="register" >
    <state id="register_form" type="form" >
        <send label="email" type="required" />
        <send label="password" type="required" />
        <send label="conf_password" type="required" />

        <transition  type="form" label="submit_register" >
            <submit target="login_page" />
            <error  target="register_error" />
        </transition>

        <onexit id="message_new_register" type="displayed?" />
    </state>

    <onentry id="section_title_signup" type="default" />
    <onexit  id="attr_email" type="attribute" />
</state>
```

Listing 3.5: Exemplo de modelação da máquina de estados

3.4 MUTAÇÕES

As mutações são casos particulares de teste que servem para melhorar a qualidade do conjunto de casos de teste e, consequentemente, do software. São introduzidos pequenos erros de utilização nos casos de teste com o objetivo de se verificar alterações no comportamento esperado do software, precavendo-se de possíveis erros que no futuro afetem a estabilidade e segurança do software.

Na análise de Reason (1990) existem três tipos de erros base, *slips*, *lapses* e *mistakes*. No entanto, a ferramenta TOM consegue ir mais longe e possibilita a introdução de erros mais específicos na utilização de uma interface gráfica. Foram então criados dois grupos de mutações, o primeiro traz-nos as mutações que estão disponíveis para se introduzir no preenchimento de formulários. No segundo grupo foram incluídas as mutações que se realizam em cliques de botões ou *links* e algumas tendências de utilização na resposta de uma página web a um pedido mais demorado.

As mutações do primeiro grupo, que vamos apresentar a seguir, encontram-se disponíveis para utilização específica pelo utilizador que está a modelar o sistema. Para tal só é necessário adicionar essa informação ao ficheiro que contém as configurações das mutações

aos formulários. Estas mutações são utilizadas automaticamente na geração de casos de teste.

- **Slips**, troca na ordem de execução das ações no formulário.
- **Lapses**, é feita a remoção de uma introdução de valores. Por exemplos, um dos campos não será preenchido.
- **Mistakes**, é realizada a modificação de um valor no campo do formulário.
- **DoubleClick**, realiza um duplo clique no botão de submissão do formulário. Quando o tempo de resposta a uma submissão de dados é demasiado demorado o utilizador tem tendência a carregar várias vezes no botão de submissão, e é que pretendemos simular para analisarmos a resposta do sistema a esta situação.

Nas mutações do segundo grupo pretendemos adicionar mais alguns erros característicos de utilização de uma página web por parte do utilizador comum. A introdução destas mutações não necessita de configuração por parte do utilizador, é um processo automatizado, sendo somente necessário indicar no TOM Generator que sejam gerados casos de teste, com este género de mutação. Os erros são os seguintes:

- **Duplo clique em elementos**, faz um duplo clique em um botão ou *link* da página web. Por vezes, quando a página web está a demorar a responder ou por outra razão, o utilizador faz duplos cliques nos elementos HTML, como botões, por exemplo. O mesmo acontece na próxima mutação.
- **Duplo clique em menus**, realiza um duplo clique nos menus de navegação.
- **Carregar no botão de voltar atrás**, normalmente utilizado em momentos em que a ligação à internet está lenta ou por engano na transição para uma página, situações em que existe alguma tendência para se carregar no botão para voltar atrás.
- **Realizar um refresh na página**, força a página a recarregar todos os dados da página web. Os motivos são muito similares aos apresentados anteriormente.

3.4.1 *Modelação das mutações no sistema*

Através do recurso à TOM Framework é possível a utilização de mutações em determinadas situações de forma automatizada, sendo que o utilizador pode definir alguns critérios de mutação, se assim o desejar, manualmente. De modo a simplificar o modelo do sistema as mutações são introduzidas na geração dos caminhos e não no modelo.

As mutações do primeiro grupo podem ser utilizadas pelo utilizador para definir certo tipos de alterações na introdução de dados na aplicação. Para tal, é necessário criar mais

um ficheiro, que segue o formato de dados da Tabela 8, que originará um ficheiro com casos de testes com a introdução destas mutações.

```
[ {
  type: "",
  model_element: "",
  value: "",
  fail: ""
}, ...]
```

type	É utilizado para representar qual o tipo de mutação que se pretende realizar. Os valores aceites para este atributo são: *lapse*, *slip*, *mistake*, *doubleClick*.
model_element	Identifica o elemento do modelo onde se introduz a mutação.
value	Este atributo só é necessário quando o tipo de mutação é *mistake*. Neste sentido, significa que o valor original do ficheiro de valores será trocado por este.
fail	Permite determinar se com a introdução desta mutação o caso de teste deverá falhar, gerando um teste inválido. Caso o valor seja definido como "1" o caso de teste irá conter falhas no teste, se o valor for "0" significa que esta mutação não irá interferir no resultado do caso de teste.

Tabela 8.: Tabela com a descrição dos elementos de mutação.

Na Listagem 3.6 podemos encontrar um exemplo de utilização das mutações no preenchimento do formulário de registo, da Listagem 3.5, seguindo a estrutura definida na tabela anterior. Por exemplo, o campo da confirmação da *password* vai sofrer uma troca de valor, o que irá originar uma falha no registo do utilizador porque as palavras passes não coincidem. Também é efetuado um duplo clique no momento do registo do utilizador mas isso não influencia o resultado final, fazendo somente a inscrição de um utilizador.

```
[   {   "type" : "mistake",
        "model_element" : "conf_password",
        "value" : "123",
        "fail" : "1"
    }, {
        "type" : "doubleClick",
        "model_element" : "submit_register",
        "fail" : "0"
    }, ... ]
```

Listing 3.6: Exemplo de configuração das mutações em ficheiro.

3.5 CONFIGURAÇÕES PARA A GERAÇÃO DE CASOS DE TESTE

Para se realizar este importante processo de geração de casos de teste, é necessário configurar o TOM Generator com alguma informação sobre a aplicação web que se pretende avaliar e também sobre como se pretende que os testes sejam gerados. A atual interface da ferramenta é ainda pouco atrativa e intuitiva, sendo necessário atribuir os valores às variáveis manualmente diretamente no código.

Algumas das principais variáveis que necessitam de ser configuradas são apresentadas na Listagem 3.7. Inicialmente definem-se os caminhos para os ficheiros que modelam a aplicação, o número de ficheiros que devem ser gerados, onde devem ser armazenados e, no final, algumas informações sobre os algoritmos de travessias que são aplicadas no grafo.

Uma variável importante e referida anteriormente é a do número de ficheiros com casos de teste gerados. Nesta variável, se o valor atribuído for 1 só será gerado um ficheiro mas sem nenhuma mutação. Se o valor variar entre 2 e 10 serão gerados ficheiros com todo o género de mutações, por esta ordem: *lapse, mistake, double click submit, remove required field, double click call, double click menu call, inject back event, inject refresh event* e mutações obtidas a partir do ficheiro de configuração.

```
/* PATH TO CONFIGURATIONS FILES */
private static final String PACKAGE_NAME = "name_of_application";
private static final String FILE_MAP = "/path_to_map_file.json";
private static final String FILE_VALUES = "/path_to_values_file.json";
private static final String FILE_MUTATIONS = "/path_to_mutation_file.json";
private static final String FILE_MODEL = "/path_to_model_file.xml";
private static final String URL = "website_url";

/* NUMBER OF GENERATED FILES TESTS & OUTPUT FOLDERS */
private static final int TEST_NUMBER = 10;
private static final String FOLDER_TESTS = "/path_to_generated_tests";
private static final String FOLDER_IMAGE_TESTS = "/path_to_generated_images";

/* INITAL & FINAL NODE, MAX VERTEX VISIT, MAX_EDGE_VISIT */
private static final String INITIAL_NODE = "root";
private static final String LAST_NODE = "last";
private static final int MAX_VERTEX_VISIT = 1;
private static final int MAX_EDGE_VISIT = 2;
private static final int N_PATHS = 1;

/* SELENIUM DEFINITIONS */
private static final int BROWSER = 0;
private static final int SCREENSHOT_ON_ERROR = 0;
private static final int BROKEN_LINKS = 0;
private static final int BROKEN_IMAGES = 0;
```

Listing 3.7: Configuração das variáveis do TOM Generator.

Existem ainda mais algumas configurações que podem ser efetuadas na ferramenta, tais como a escolha do algoritmo que realiza a travessia sobre o grafo, a escolha do *browser* (Google Chrome, Firefox, Opera, Safari) onde são realizados os testes, definir que quando ocorre uma falha no teste é automaticamente capturada um *screenshot* da página web onde o teste falhou e ativar uma ação que verifica se todos os *links* para outras página e imagens se encontram válidos ao entrar em cada página web ou cada estado.

3.6 GERAÇÃO DE CASOS DE TESTE

De acordo com o IEEE Standard 610 (IEEE, 1990) um caso de teste é constituído por um conjunto de testes de entrada, condições de execução e obtenção dos resultados para comparar com o domínio especificado, de forma a verificar se os requisitos definidos estão a ser cumpridos.

Depois da leitura e interpretação dos ficheiros que modelam o sistema em teste, o TOM Generator realiza internamente várias etapas até à geração de casos de teste executáveis. Inicialmente o modelo do sistema é transformado num grafo com toda a informação do sistema. Após esta etapa, e com no base no grafo obtido é aplicado um algoritmo de travessias dando origem a casos de teste abstratos, que por fim são utilizados para gerar os casos de teste executáveis.

Para se realizar a transformação do modelo em um grafo, recorre-se a um multigrafo. Este, permite a criação de ciclos e possui arestas em que o vértice de origem é o mesmo de destino. Este tipo de grafo é necesssário visto que numa aplicação web existem eventos ou pedidos assíncronos que retornam para a mesma página. Como referido por Rodrigues (2015) o modelo do sistema passa por um parser que o transforma num grafo, para depois se aplicar um algoritmo de travessia sobre a estrutura criada. No grafo obtido os vértices representam páginas web e as arestas representam as ações ou interações do utilizador na interface gráfica.

Através da aplicação de um algoritmo de travessia sobre o grafo, são gerados caminhos que representam sequências de ações que vão ser executadas sob a interface gráfica, obtendo-se assim casos de teste abstratos, independentes da linguagem. É também nesta fase que são também gerados os casos de teste abstratos onde foram introduzidas as

mutações automáticas.

No TOM Generator o utilizador tem a possibilidade de escolher qual o algoritmo de travessia que pretende utilizar para gerar os caminhos abstratos. Cada um dos algoritmos têm particularidades diferentes, originando casos de testes diferentes. Os algoritmos e os seus critérios de cobertura são os seguintes:

- **AllPaths** - Procura todos os caminhos existentes com base no nodo inicial e no número máximo de passagens por vértices e arestas do grafo.
- **BellmanFordShortest** - Procura o caminho mais curto entre o nodo inicial e final, sendo possível restringir o número de arestas máximo pelo qual se passa.
- **KShortestPaths** - Algoritmo que determina os k caminhos mais curtos em ordem crescente de tamanho entre um nodo inicial e final, sendo k o número máximo de arestas a visitar.

Na última etapa são gerados os ficheiros com os casos de teste concretos, a realizar na interface gráfica do *browser*, baseados nos caminhos obtidos no processo anterior. A ferramenta começa por converter os caminhos abstratos em código *JAVA*, estruturando as ações em instruções *Selenium*, com recurso a anotações de *TestNG*.

Na Listagem 3.8, é apresentado um exemplo da estrutura de uma parte do código gerado para a validação de um elemento na página web. É realizada uma verificação de verdade da visualização de um botão na interface gráfica e caso esta afirmação não seja verdade, é lançada uma exceção com uma descrição do erro detectado para ajudar o utilizador na avaliação dos testes.

```
@Test(invocationCount = 1, groups = "1")
public void path1_test015() throws IOException, InterruptedException {
    ...
    Reporter.log("␣Enter␣in␣page␣ontology_new_show␣<br>");
    try {
        assertTrue(driver.findElement(By.cssSelector("button.btn.btn-ar.btn-warning"))
            .isDisplayed());
        Reporter.log("Element␣displayed?␣" + driver.findElement(By.cssSelector(".btn-
            ar.btn-warning")).isDisplayed());
    } catch (AssertionError _x) {
        Reporter.log("[Fail]:␣In␣a␣Validation␣new_ontology_link");
        fail("␣[Fail]:␣In␣a␣Validation␣new_ontology_link␣[Message]␣=>␣" + _x.
            getMessage() );
    }
    ...
}
```

Listing 3.8: Exemplo da estrutura do código executável gerado.

No Capítulo 6 vamos analisar a aplicação da *framework* a uma aplicação web, onde será demonstrado e debatido a execução dos casos de teste na interface gráfica.

3.7 CONCLUSÕES

Neste capítulo foram apresentadas as capacidades de utilização da *framework* TOM. O TOM Generator é uma ferramenta capaz de gerar e executar múltiplos casos de teste a aplicações web, com a introdução de mutações.

No decorrer do processo de análise do código do TOM Generator, foram atualizadas e removidas algumas dependências do projeto, o que permitiu adicionar à ferramenta a capacidade de criar testes para outros browsers, tais como o Safari e Opera. Foram ainda removidos ficheiros dos pacotes que não estavam a ser utilizados.

4

IRIT: GERAÇÃO DE CENÁRIOS

Neste capítulo é apresentado um exemplo de flexibilidade e capacidade de adaptação da *framework* a um caso de estudo proposto pelo *Institut de Recherche en Informatique de Toulouse (IRIT)*. O objetivo principal desta abordagem é a geração de cenários de teste com base em modelos de tarefas HAMSTERS[1].

Foi necessário adaptar o TOM Generator para trabalhar a partir de modelos de tarefas. Para tal, desenvolveu-se um algoritmo que traduz os modelos de tarefas em máquinas de estado. Este módulo ainda não se encontra implementado na *framework*, daí que se tenha realizado este processo de forma manual. Obtida a máquina de estados e realizada a leitura na ferramenta, implementou-se um novo módulo para tornar os casos de teste abstratos em cenários de teste concretos na linguagem que o grupo de investigação pediu.

O trabalho desenvolvido nesta parceria permitiu ainda a colaboração na escrita de um artigo científico (Campos et al., 2016), onde se demonstrou e validou a capacidade de geração de cenários para análise de desvios de comportamentos nos modelos de tarefas.

4.1 CASO DE ESTUDO

Como referido por Martinie et al. (2015), o HAMSTERS é uma ferramenta utilizada na modelação de tarefas de utilizadores, apoiado por notações que representam as interações humanas de forma hierárquica e ordenada. No mais alto nível de abstração, é possível dividir os objetivos em sub-objetivos, onde cada um destes últimos pode ser dividido em atividades. O resultado final é uma árvore, onde cada nó representa uma tarefa ou um operador temporal, como ilustrado no exemplo da Figura 5.

O caso de estudo desenvolvido pelo IRIT tem como domínio de aplicação a área da aviação e representa um caso específico de segurança e missão crítica. A unidade de con-

1 https://www.irit.fr/recherches/ICS/softwares/hamsters/ (visitado pela última vez em 25/10/2016).

trolo de voo (*Flight Control Unit (FCU)*), nos *cockpits* interativos, é composta por um painel de hardware com vários dispositivos eletrónicos, como botões e ecrãs que permitem que os membros da tripulação configurem os ecrãs de voo e navegação. O FCU Software é considerado uma interface gráfica interativa que pretende substituir as unidades hardware FCU, e que está dividida em duas secções interativas:

- **EFIS_CP** - É o painel de controlo do sistema eletrônico de informação de voo, que permite configurar a pilotagem e os ecrãs de navegação.
- **AFS_CP** - Painel de controlo do sistema de voo automático, utilizado para configurar o estado e parâmetros do piloto automático.

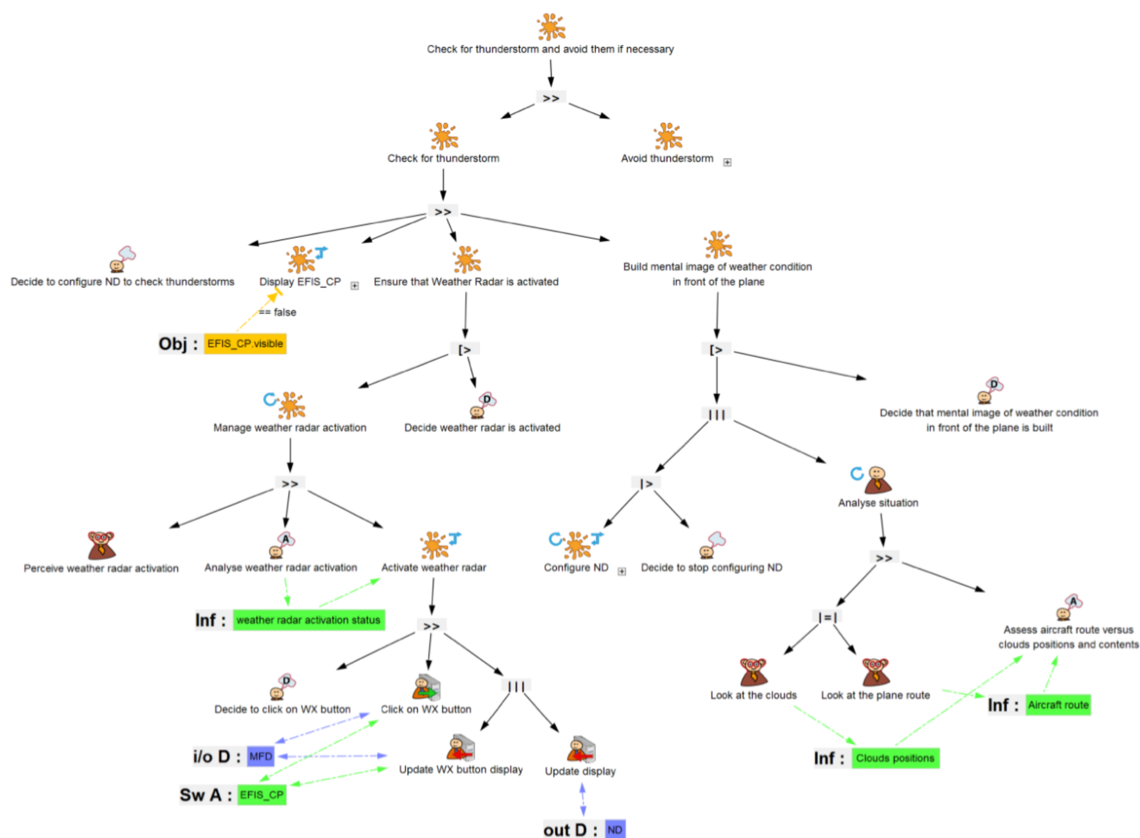


Figura 5.: Modelo de tarefas HAMSTERS para a tarefa principal *Check for thunderstorm and avoid them if necessary*.

O painel de controlo EFIS, representado na Figura 6, apresenta-nos duas interfaces, com e sem a ativação do radar meteorológico. O modelo de tarefas apresentado na Figura 5 corresponde às diferentes possibilidades para verificar se existem tempestades na rota do avião e evitá-las, se necessário. Este modelo foi construído com base nas interações possíveis com a interface gráfica da Figura 6, sendo que para o caso de estudo atual só nos iremos focar

na parte em que o piloto verifica se há uma tempestade (tarefa *check for thunderstorm* da Figura 5).

Seguindo o modelo de tarefas apresentado, para se verificar se uma tempestade (tarefa *check for thunderstorm*) vai aparecer na rota do avião, o piloto deve verificar se o ecrã EFIS_CP está visível e se o radar meteorológico está ativo (tarefa *ensure that weather radar is activated*), caso não esteja ativo é necessário fazer as interações necessárias com a interface até o radar ficar ativo. Posteriormente é necessário que o piloto faça uma análise das condições meteorológicas (tarefa *build mental image of weather condition in front of the plane*), configurando o ecrã de navegação (*configure ND*) enquanto analisa a situação.

Quando o piloto decide que tem uma imagem válida das condições climáticas (tarefa *Decide that mental image of weather condition in front of the plane is built*), é então necessário decidir se a rota do avião está correta ou deve ser alterada (tarefa *Avoid thunderstorm*).

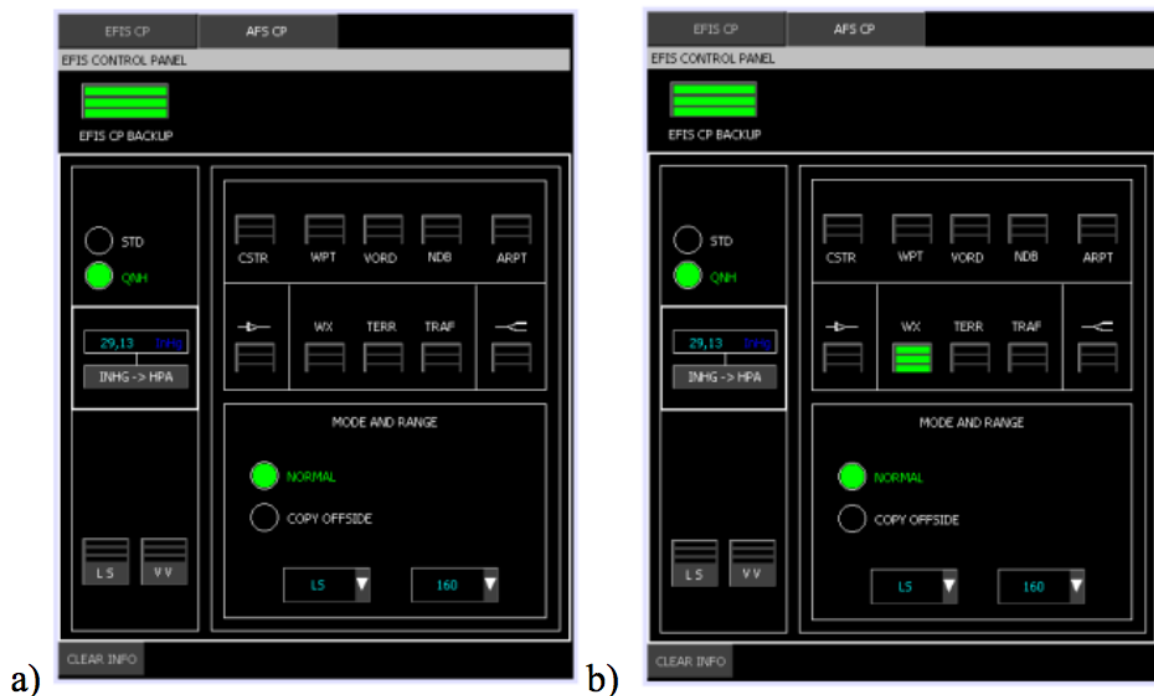


Figura 6.: Painel de controlo EFIS, com b) e sem a) ativação do radar metereológico.

4.2 ARQUITETURA

O TOM Generator foi implementado numa estrutura modular, desta forma, permitiu que grande parte dos módulos fossem aproveitados na leitura da máquina de estados e na

geração dos cenários de teste abstratos. Foi necessário implementar um módulo para transformar os cenários abstratos em cenários concretos. Na arquitetura da Figura 7 vemos o módulo que foi desenvolvido, IRITGen, e os restantes que foram reutilizados.

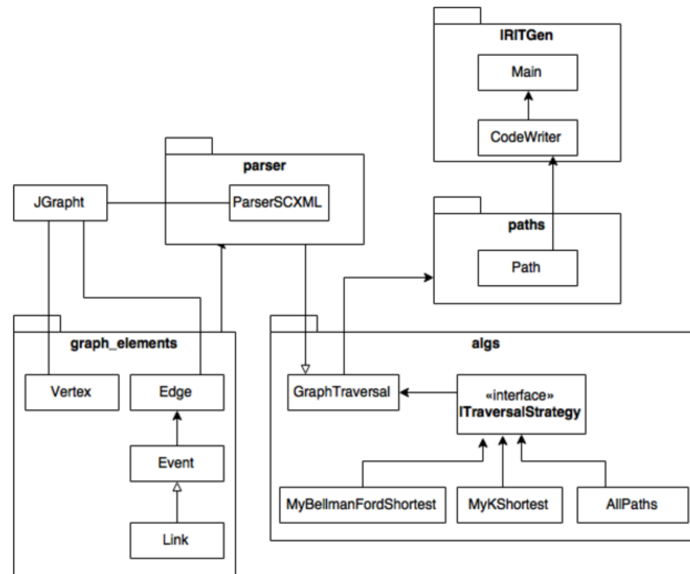


Figura 7.: Arquitetura da geração de cenários

4.3 CONVERSÃO DO MODELO DE TAREFAS

Para se conseguir realizar a geração de cenários é necessário converter o modelo de tarefas numa máquina de estados. Para tal, define-se cada estado da máquina como o conjunto de tarefas possíveis em cada momento da interacção. Nesta fase o modelo foi construído manualmente com recurso ao simulador da ferramenta HAMSTERS (ver Figura 8) anotando, em cada etapa da simulação, quais as tarefas possíveis. Desta forma, a implementação do módulo de criação da máquina de estados terá de ser realizado pelo IRIT recorrendo às capacidades de simulação já existentes na ferramenta.

A máquina de estados segue um formato muito similar ao explicado na Secção 3.1, sendo que para o caso de estudo atual foram só utilizados os elementos *state* e *transition*. O elemento state contém o atributo *id*, que serve de identificação do estado. O elemento *transition* é composto por 2 atributos, o *id* e o *target*. No primeiro, identificamos o nome da tarefa, que corresponde ao modelo de tarefas. No atributo *target* identifica-se o estado para o qual se vai transitar, se for definida aquela ação.

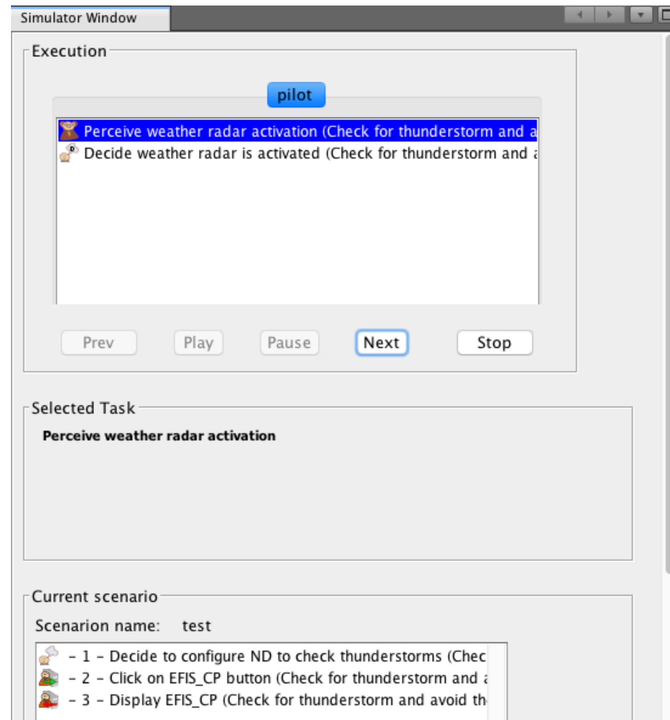


Figura 8.: *Screenshot* da ferramenta HAMSTERS em simulação.

Com informação obtida da simulação construiu-se uma máquina de estados que contemplava todas as transições existentes entre as tarefas, como se apresenta na Listagem 4.1. Como é possível perceber pela listagem e pela Figura 8, quando o piloto se encontra no estado "2" é possível executar duas ações. A primeira ação ("perceive_weather_radar_activation") dá acesso ao estado "3". Se for seleccionada a outra opção ("decide_weather_radar_is_activated") então a máquina de estados avança para outro estado ("9").

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" initialstate="1">
    <state id="1">
        <transition id="decide_to_configure_nd_to_check" target="2"/>
    </state>
    <state id="2">
        <transition id="perceive_weather_radar_activation" target="3" />
        <transition id="decide_weather_radar_is_activated" target="9" />
    </state>
    <state id="3">
        <transition id="analyse_weather_radar_activation" target="4" />
        <transition id="decide_weather_radar_is_activated" target="9" />
    </state>
    ...
</scxml>
```

Listing 4.1: Exemplo da máquina de estados convertida.

4.4 GERAÇÃO DOS CENÁRIOS

Depois de se ter escolhido o algoritmo de travessias e obtido os caminhos com os casos de teste abstratos foi necessário transformar estes testes em cenários concretos. Os cenários de teste gerados, como o exemplo da Listagem 4.2, contém uma lista de etapas a percorrer durante a simulação que ao serem co-executadas com o modelo de tarefas detectam se houve algum desvio no comportamento esperado. Quando existe algum comportamento inesperado ou que não foi executado, estamos perante uma falha.

Para a geração dos cenários foram utilizados os três tipos de algoritmos (*AllPaths, MyKShortestPaths e MyBellmanFordShortestPath*) implementados na ferramenta. O primeiro algoritmo procura todos os caminhos existentes no grafo gerado, tendo sido gerados 9601 cenários de teste. O algoritmo *MyKShortestPaths* procurou os caminhos existentes entre dois nodos, o inicial e final, do grafo gerado, obtendo-se 1176 cenários de teste. Por último, foi aplicado o algoritmo de *MyBellmanFordShortestPath* onde só se obtém um único cenário de teste, o mais curto entre o nodo inicial e final.

```
<?xml version="1.0" encoding="UTF-8"?>
<hamsterscenario date="0" simulatedmodel="Check␣for␣thunderstorm␣and␣avoid␣them␣if␣
    necessary" version="2">
    <objects/>
    <steps>
    <step referencemodel="Check␣for␣thunderstorm␣and␣avoid␣them␣if␣necessary" role="
        pilot" taskdate="0" taskdatelong="0">
        <task taskid="t17"/>
    </step>
    <step referencemodel="Check␣for␣thunderstorm␣and␣avoid␣them␣if␣necessary" role="
        pilot" taskdate="1" taskdatelong="1">
        <task taskid="t90"/>
    </step>
    <step referencemodel="Check␣for␣thunderstorm␣and␣avoid␣them␣if␣necessary" role="
        pilot" taskdate="Tue␣Jan␣05␣14:51:54␣CET␣2016" taskdatelong="1">
        <task taskid="t183"/>
    </step>
    ...
    </steps>
</hamsterscenario>
```

Listing 4.2: Parte do cenário gerado.

Para o IRIT, foram enviados cerca de 100 cenários de teste, gerados a partir do algoritmo de *MyKShortestPaths*, para co-execução.

4.5 CONCLUSÕES

A co-execução de cenários gerados a partir do modelo de tarefas permitiu demonstrar a viabilidade da detecção de erros. Isto acontece, por exemplo, quando durante a co-execução dos cenários e da aplicação se detectam tarefas acessíveis no modelo, quando na verdade isso não é possível na aplicação. A detecção dos erros permite melhorias e alterações, quer na aplicação, quer no modelo de tarefas construídos.

A atual forma de geração de cenários é ainda bastante simplista, sendo que em futuras iterações do projeto com o IRIT é necessário adicionar suporte para as mutações mais genéricas, como os *lapses*, *slips* e *mistakes*, na geração dos cenários de teste. Além disso, é também essencial suportar tipos de dados na máquina de estados, para mais tarde estes dados serem enviados no decorrer da execução dos cenários.

Apesar de terem sido gerados mais de 1000 cenários de teste, co-executar tal número de cenários pode não ser realista. Assim, torna-se necessário que este número de casos de teste seja reduzido para que o processo de execução não seja tão longo. Uma das primeiras situações que ajudará na redução de cenários será não serem gerados ficheiros com cenários repetidos. Outras soluções passariam pela imposição de novas regras na construção da máquina de estados.

5

TOM EDITOR

5.1 ABORDAGEM PROPOSTA

A criação dos ficheiros de configuração de uma interface gráfica era até ao momento um processo completamente manual, pelo que utilizador perdia demasiado tempo a configurar o modelo e os restantes dados, existindo ainda o risco de introdução de erros no modelo. À medida que o utilizador vai adicionando propriedades sobre a máquina de estados da interface gráfica aos ficheiros, estes acabam por começar a ficar confusos, grandes, e com maiores probabilidades de problemas ao nível de ligação entre cada ficheiro.

De modo a ultrapassar este último problema, decidiu-se criar uma aplicação que permitisse definir o modelo da máquina de estado e restantes configurações de forma mais intuitiva, rápida e automatizada (o TOM Editor). Assim, a aplicação estará preparada para conseguir gravar a interação do utilizador com a página web que se pretende testar, fazendo automaticamente a criação do modelo do sistema e do mapeamento da interface gráfica. A qualquer momento o utilizador pode exportar os ficheiros nos formatos específicos para leitura no TOM Generator.

Desenvolveu-se um *plugin* para um *browser*, com o objetivo de se capturar a interação do utilizador com a página web sobre teste e conseguir, ao mesmo tempo, definir um modelo do sistema com base na interação do utilizador. O modelo é apresentado ao utilizador numa janela onde tem à sua disposição diversas funcionalidades que lhe permitem adicionar, alterar e/ou remover informações do sistema.

A implementação da aplicação foi realizado em 5 fases. Na primeira fase, foi realizada uma análise ao formato de dados dos ficheiros de entrada no TOM Generator. Posteriormente, construiu-se o modelo de dados necessário para a aplicação ser implementada e elaboraram-se os mockups que definiam a interface gráfica da aplicação criada. De seguida, e antes de o desenvolvimento da extensão começar, escolheram-se e validaram-se as tecnologias em que a aplicação deveria ser implementada. A terceira fase foi marcada pelo início

do desenvolvimento da interface gráfica e pela criação de funcionalidades que permitam ao utilizador criar, alterar e remover dados do modelo através de formulários. Na quarta fase foi estabelecida a ligação entre a aplicação e a página web que se está a avaliar. Passou a ser possível ao utilizador que está a adicionar ou alterar informação do modelo seleccionar um elemento da DOM para obter o seu mapeamento na página. Por último, foi adicionada à extensão a capacidade de obter a interação que o utilizador tem com a página, facilitando e automatizando assim a criação do modelo do sistema.

Nas próximas secções vamos discutir os principais aspectos de cada fase.

5.2 ARQUITETURA DA SOLUÇÃO

Depois da análise do formato dos ficheiros de configuração, a estrutura definida para a aplicação acaba por ficar bastante simples, permitindo armazenar a informação sobre a interação do utilizador com a página e as validações, fundamentais para testar a interface.

Cada estado representa uma página web, logo uma aplicação web pode ser definida como um conjunto de estados. Cada estado é caracterizado por inúmeras ações. Estas ações podem ser validações, transições e formulários. Os formulários servem para descrever a introdução de dados na página e por isso tem a particularidade de ainda poderem virem a sofrer alguma mutação. As transições asseguram a ligação de uma página para outra, isto é, de um estado para outro.

O mapeamento assegura a ligação entre as ações do modelo e a aplicação. O mapeamento guarda as informações de como alcançar determinado elemento na página web, sendo um elemento fulcral na geração dos casos de teste.

O modelo de domínio que representa esta estrutura é representado na Figura 9.

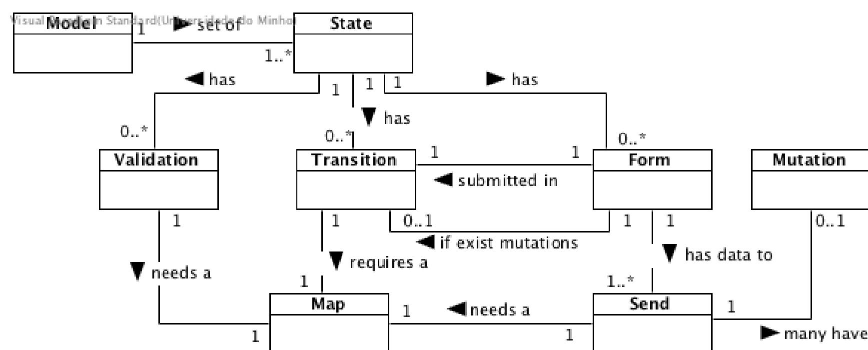


Figura 9.: Modelo de domínio do TOM Editor

Concluída a modelação estrutural, procedeu-se à elaboração de alguns protótipos de baixa fidelidade, em papel, que ajudaram a fornecer uma ideia aproximada do aspecto de algumas funcionalidades da aplicação. Seleccionadas as melhores ideias, procedeu-se à elaboração de alguns *mockups*, com recurso ao *balsamiq*[1], onde se obteve um aspecto aproximado da interface gráfica da aplicação.

Na Figura 10 são apresentados dois *mockups* que representam o ambiente geral do editor da aplicação. São visíveis algumas diferenças entre os mockups, nomeadamente na forma como são apresentados os estados, num modelo através de várias caixas e no outro com recurso a um acordeão de caixas. Outra diferença visual que se pode encontrar é na forma como é apresentada a informação sobre cada ação, nomeadamente com o local onde se deve encontrar esta barra de informação.

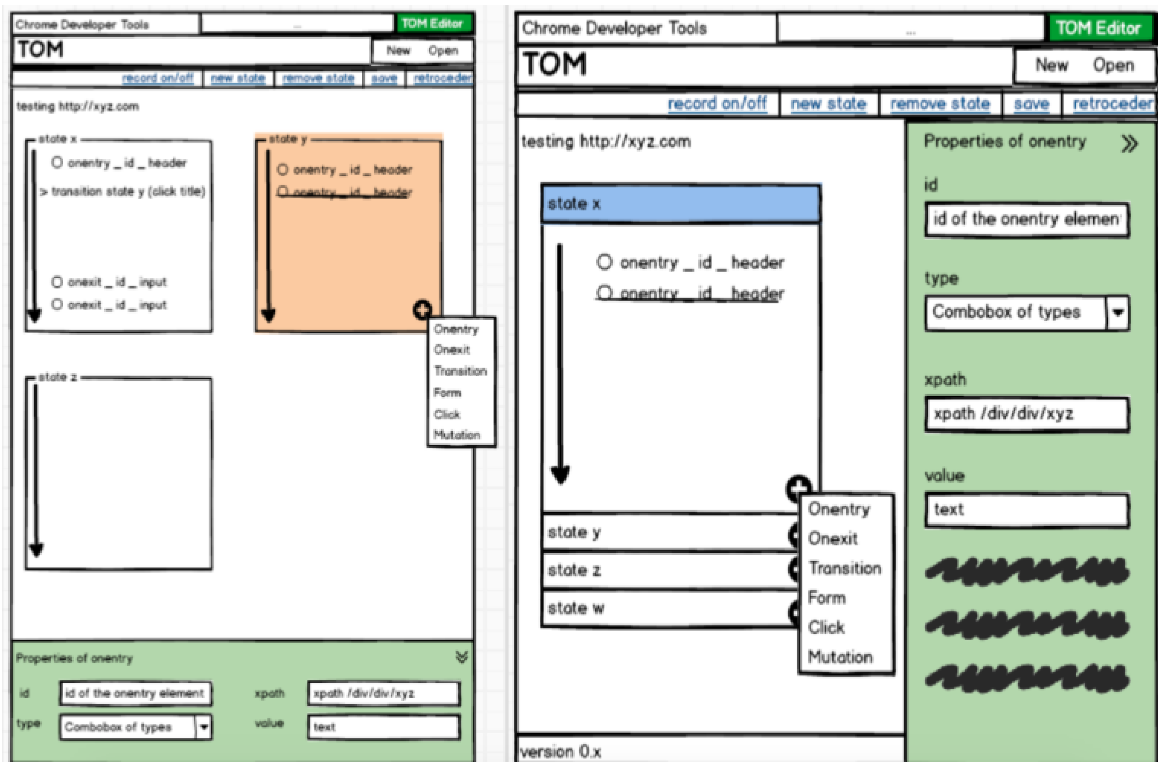


Figura 10.: Mockups da interface do TOM Editor

Através da análise das diferenças entre os *mockups* e com a recolha de algumas apreciações de utilizadores sobre a interface, foi possível perceber que a utilização de múltiplas caixas no editor era preferível à utilização do acordeão, porque o utilizador tem acesso mais fácil e rápido ao modelo que constitui o sistema. Ficou ainda decidido que a barra de informação das ações se iria situar na lateral direita da aplicação.

1 https://balsamiq.com (visitado pela última vez em 10/11/2016).

5.3 TECNOLOGIAS E LIMITAÇÕES

Atualmente, já existem vários *plugins* (extensões para os *browsers*) disponíveis, em parte referidos na Secção 2.6.4, que permitem a captura, gravação e execução de interações do utilizadores com as aplicações web. Daí que, e tendo em conta os objetivos desta dissertação, a melhor solução de implementação para o nosso problema, de construções de modelos do sistema, passou pela construção de uma extensão para o browser.

Após uma investigação às *Application Programming Interface (API)* públicas de desenvolvimento de extensões dos principais *browsers* do mercado, concluiu-se que neste momento iria ser muito complicado implementar o projeto em todos os browsers devido às atuais diferenças entre as APIs do Google Chrome, Safari e Firefox e à necessidade de aprender a trabalhar com cada uma delas. No entanto, encontrou-se uma solução intermédia, Crossrider, onde através da sua enorme API era possível construir uma extensão suportada pelos principais *browsers*. Na exploração desta *framework* encontramos alguns entraves à implementação da nossa extensão, como a comunicação entre a página web e o TOM Editor, a comunidade de suporte ser bastante reduzida e a necessidade de recorrer a vários *plugins* internos para suportar o projeto. Mais tarde, a decisão de não se continuar o projeto nesta *framework* veio a revelar-se a mais correta, porque a mesma veio a ser descontinuada no final de Setembro de 2016.

Optou-se por desenvolver uma extensão no Google Chrome, que nos oferece uma API[2] bastante completa e documentada, configuração do projeto simples, rápida e com suporte para tecnologias de *front-end*. Para o género de extensão que se pretende construir, o Google Chrome assegura pelo menos três possibilidades para a localização da interface:

1. **PÁGINA POPUP**

   Na Figura 11, é apresentado um exemplo de uma interface *popup* de uma extensão que aparece sobre a página web. No entanto, quando a *popup* desaparece ele perde o todo o seu estado, isto é, todos os dados que foram inseridos através da janela são perdidos.

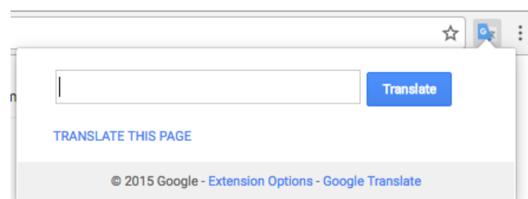


Figura 11.: Interface gráfica da janela de popup das extensões no Google Chrome.

2 https://developer.chrome.com/extensions (visitado pela última vez em 05/01/2017).

2. PÁGINA DAS APLICAÇÕES

Como vemos na Figura 12 o chrome tem uma área destinada a aplicações mais completas e que não necessitam de ter comunicação com outras páginas do *browser*.

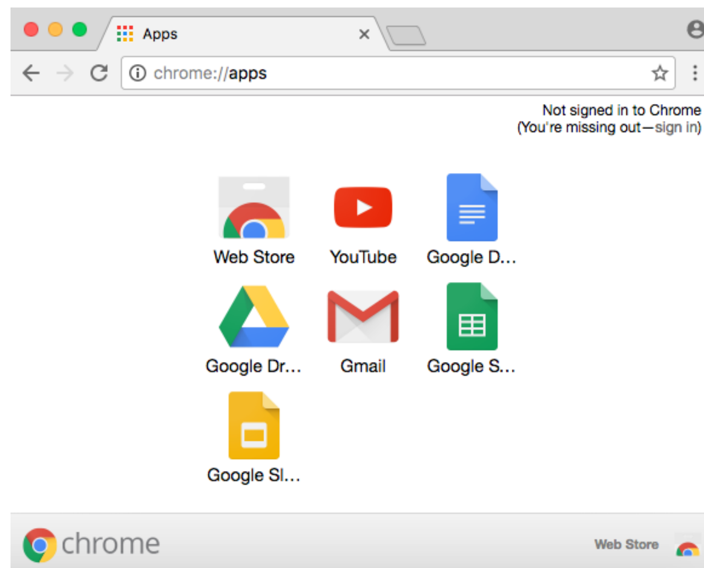


Figura 12.: Interface gráfica da página de aplicações do Google Chrome.

3. PAINEL NA JANELA DE INSPEÇÃO DO BROWSER

Como é possível verificar na Figura 13, é possível criar novos painéis na janela de inspeção do browser. Esta janela pode ser aberta a partir de qualquer *tab* no *browser*, ficando assim somente associada a essa página.

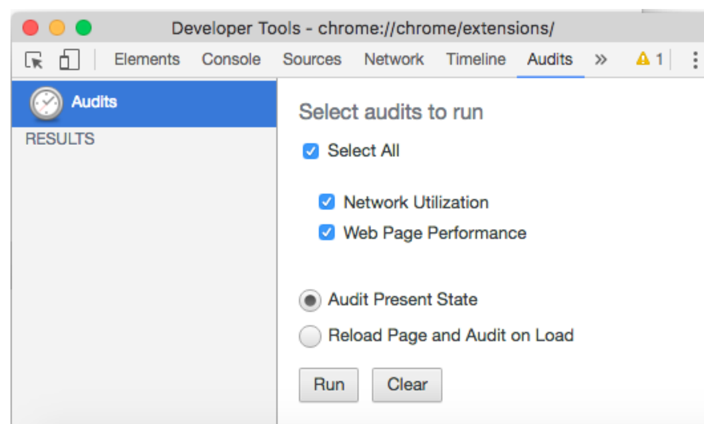


Figura 13.: Interface gráfica da janela de inspeção do browser no Google Chrome.

Acabamos por escolher a terceira opção (Figura 13) por ser aquela que se mantém mais próxima da página que se está a testar, onde será mais fácil visualizar as interações que o

utilizador fez e onde normalmente estão as ferramentas utilizadas pelos *developers* no desenvolvimento das aplicações.

Como referido anteriormente, as extensões no Google Chrome podem ser implementadas com recurso a tecnologias de *front-end*, como AngularJS, Ember.JS, Backbone.JS. Isto permite-nos tirar o máximo partido na utilização de alguma destas tecnologias, como a construção estruturada de código JS, reutilização de código, manutenção mais fácil e uso do padrão arquitetural *Model-View Controller (MVC)* na implementação do projeto. Neste sentido, optou-se por escolher Angular 2, uma das mais recente tecnologias da Google, como base de desenvolvimento do projeto.

Angular 2[3] é uma framework orientada ao desenvolvimento de aplicações de alto desempenho na camada de apresentação, com uma boa capacidade de renderização de HTML e CSS, sem necessidade de instalação e modular. Devido à utilização de *Typescript (TS)*[4] como linguagem principal na *framework*, é possível escrever um código mais modular e escalável, com recurso a tipos de dados, classes e interfaces. Um dos pré-requisitos para a utilização desta tecnologia é a necessidade de Node.js[5] na base da aplicação para conseguir correr, testar, criar *builds* da extensão e ter acesso aberto a milhares de "bibliotecas" que se integram com a aplicação. Para gerir o acesso a estas "bibliotecas" utilizou-se a NPM[6]. De realçar a utilização de Gulp[7], uma "biblioteca", que me permitiu gerir e automatizar o *workflow* de geração de *builds*. A biblioteca "*gulp-trypescript*'' é responsável por transformar o código TS em JS. Este processo é designado por *transpile*, onde uma linguagem de programação é transformada em outra equivalente, com o mesmo significado.

A curva de aprendizagem da *framework* foi bastante acentuada no início, por ser uma experiência completamente nova na utilização destas tecnologias, dada a quantidade de novos conceitos que foram necessários aprender e porque na altura da implementação do projeto o Angular 2 ainda se encontrar numa versão "*release candidate*".

Na fase inicial do projeto, foram gastas várias horas até se conseguir ter uma base estável da aplicação a correr como uma extensão no *browser*. Foi necessário configurar o manifesto da extensão (*manifest.json*[8]) com algumas definições após a leitura da API e configurar o transpile de TS para que o processo não fosse realizado pelo *browser*, isto é, pelo *engine* de conversão em JS do Google Chrome, e antes fosse utilizada a biblioteca "*gulp-typescript*" no

3 https://angular.io/ (visitado pela última vez em 11/12/2016).
4 https://www.typescriptlang.org/ (visitado pela última vez em 11/12/2016).
5 https://nodejs.org/ (visitado pela última vez em 11/12/2016).
6 https://www.npmjs.com/ (visitado pela última vez em 11/12/2016).
7 http://gulpjs.com/ (visitado pela última vez em 11/12/2016).
8 https://developer.chrome.com/extensions/manifest (visitado pela última vez em 15/12/2016).

decorrer do processo de criação da build.

O manifesto da extensão é constituído por várias variáveis, em formato JSON, que descrevem (*name, description, version*) e definem a extensão (as seguintes). Algumas das variáveis indicam onde devem ser encontrados os módulos que constituem a extensão, tais como o *background*, *content_scripts* e *devtools_page*. A variável *permissions* indica quais os recursos do browser a que aplicação quer ter acesso e a *content_security_policy* indica as políticas de segurança definidas para a aplicação.

A estrutura atual de dados é armazenada na memória do browser enquanto o utilizador mantiver aberta a sessão da janela de inspeção, sendo que após o utilizador fechar esta janela todos os dados são perdidos. Neste momento, ainda não se está a tirar partido da base de dados que o *browser* dispõe ou da *cache*, mas se o utilizador pretender guardar o modelo, é exportado um ficheiro com toda a informação introduzida no TOM Editor.

5.4 COMUNICAÇÃO ENTRE APLICAÇÃO E O BROWSER

Um dos elementos mais importantes e que asseguram o acesso completo a todas as funcionalidades do TOM Editor é a ligação existente entre a aplicação e o Google Chrome. A comunicação é assegurada por uma ligação de longa duração até ao fecho da aplicação, com recurso às propriedades da API chrome.runtime[9] para o envio de mensagens entre as várias componentes da extensão. Antes de se ter tomado a decisão de usar este tipo de ligação, analisaram-se os tipos de ligações possíveis na API do Google Chrome.

São três as componentes que compõem a comunicação na extensão, ver Figura 14. O *"connection service"* é o componente responsável por criar e enviar as mensagens com os pedidos de ativação/desativação da captura automática da interação do utilizador com a página web, a seleção de elementos da DOM. Este componente realiza ainda o tratamentos dos dados recebidos da componente *"background page"*. A componente *"background page"* é muito similar a um *handler* porque trata de receber e re-enviar as mensagens para os componentes corretos. Por último, a componente *"content scripts"* trata de responder aos pedidos criados no *"connection service"*, ao injectar na camada do DOM da página web eventos que acionam scripts para determinadas ações.

9 https://developer.chrome.com/extensions/runtime (visitado pela última vez em 15/12/2016).

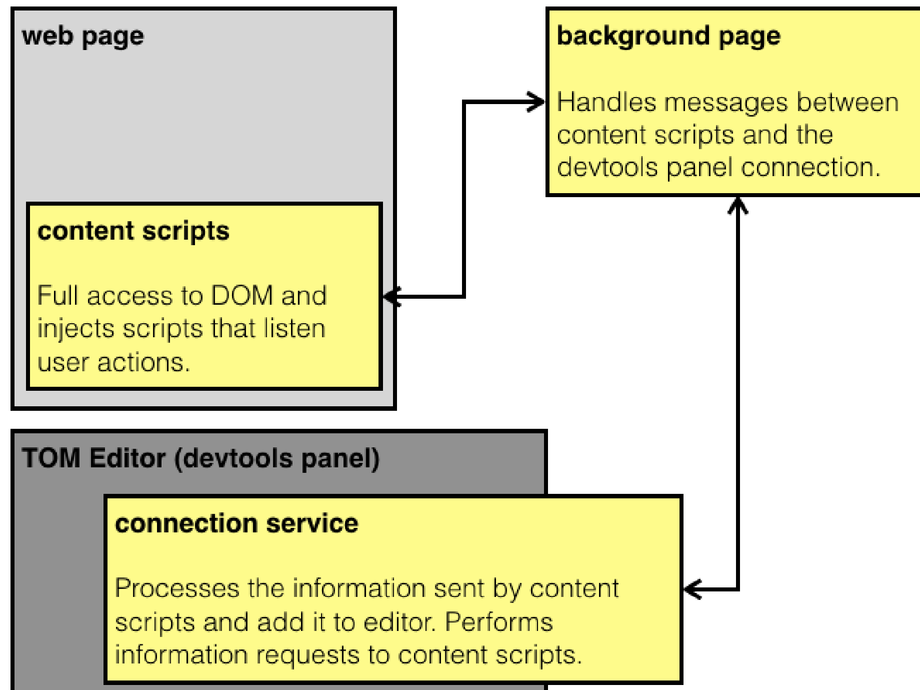


Figura 14.: Comunicação por componentes entre o TOM Editor e o *browser*.

A comunicação entre os componentes da extensão é baseada em duas portas (chrome. runtime.Port[10]), uma no componente do *"connection service"* e outra no componente *"content scripts"*. Quando o TOM Editor é aberto na janela de inspeção do *browser* é criada e aberta uma porta no componente *"connection service"*. De seguida, é enviado a partir deste componente uma mensagem para o *"background page"*, onde é dado conhecimento dos dados da porta aberta. A *"background page"* que está permanentemente à escuta das mensagens vindas do *"connection service"*, armazena os dados desta porta e envia logo depois, para o componente *"content scripts"*, uma mensagem única (*one-time requests*) a pedir conhecimento da porta criada no componente. Ao receber o pedido, o componente *"content scripts"*, abre a sua própria porta e comunica-o por mensagem ao *"background page"*, que a armazena. Nesta altura a comunicação interna da extensão está estabelecida e pronta a ser utilizada, através destes componentes.

A comunicação entre os componentes assume um papel bastante crucial para o sucesso dos objetivos definidos nesta dissertação. A forma de comunicação anteriormente descrita é a que nos parece mais correta para a comunicação entre as componentes. Foram analisados outras formas de implementação, através do recurso *one-time requests* mas não se obteve os resultados esperados.

10 https://developer.chrome.com/extensions/runtime#type-Port (visitado pela última vez em 15/12/2016).

As mensagens enviadas em cada pedido ou resposta seguem uma estrutura própria, definida na Listagem 5.1, com o seguinte formato:

- **type:** identifica o tipo de mensagem que se vai enviar, como por exemplo, o pedido de inicialização de comunicação da aplicação, de seleção de um elemento, captura da interação do utilizador, resposta a algum pedido, etc...
- **data:** onde se armazenam os valores que se pretende enviar entre as camadas da extensão. É um atributo opcional.
- **response:** este parâmetro é similar ao *type* e só é utilizado quando se trata de uma mensagem de resposta a algum pedido, colocando o tipo do pedido neste atributo, e alterado o valor do *type* para resposta.
- **source:** este atributo é utilizado para se identificar qual o componente que está ativo quando se estão a adicionar novos dados ao modelo (validação, transição ou formulário).

```
export interface Message<T> {
    type: MessageType;
    data?: T;
    response?: MessageType;
    source?: MessageSource;
}
```

Listing 5.1: Formato da estrutura da mensagem de comunicação.

5.5 FUNCIONALIDADES DA EXTENSÃO

Para além de permitir capturar automaticamente a interação do utilizador com a página web, o TOM Editor permite, caso necessário, adicionar novas ações e estados ao modelo de forma manual. Existem disponíveis várias funcionalidades no modo edição, como a adição de estados, transições, validações e formulários.

As ações estão associadas a um estado. Para adicionar, alterar ou remover uma ação é necessário recorrer aos diferentes tipos de formulários existentes, um para cada tipo de ação. Se a informação no formulário estiver bem preenchida é possível adicionar a informação ao modelo. Na Figura 15, podemos visualizar os menus de adição de informação das ações aos estados.

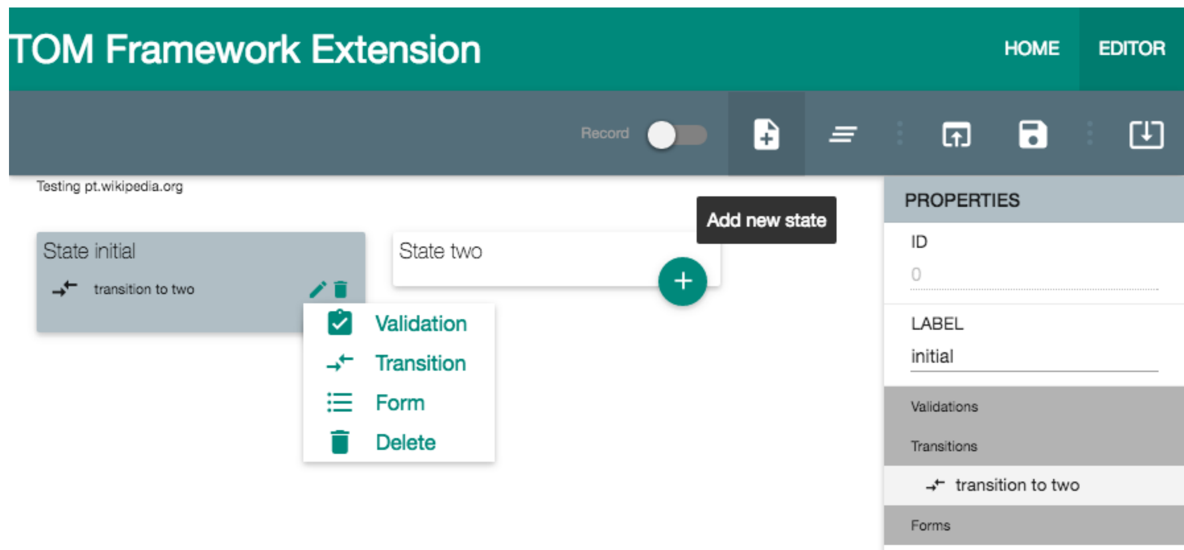


Figura 15.: Menus de adição de ações e estados do TOM Editor.

Figura 16.: Formulário de adição de validações do TOM Editor.

A solução para a adição de informação encontra-se dividida em duas partes, ver Figura 16. A primeira é referente aos dados do tipo de ação que vai ser adicionada, sendo que na segunda parte o utilizador precisa de mapear a ação com um elemento HTML da DOM (mais detalhes na próxima subsecção). No caso de ações de validação é obrigatório identificar se é uma validação de entrada ou de saída, o tipo de teste que vai ser efetuado, conforme explicado na Secção 3.3, e dependendo da escolha do caso anterior, é necessário

adicionar um valor resultante expectável. Para se criar uma transição, basta identificar qual o estado para o qual se vai transitar ou criar, caso não exista, um novo estado. As ações do tipo formulário são mais complexas, porque é necessário definir inicialmente qual o estado para o qual se vai transitar depois de se submeter o mesmo na página web, e ainda, se houver mutações, definir as informações desta. Depois deste passo concluído, é possível adicionar os dados que vão ser enviados para a página sobre teste, sendo possível enviar o tipo de dados do elemento *send*, explicado na Secção 3.1.

5.5.1 *Selecção do elemento na DOM*

Como visto anteriormente, o utilizador necessita de definir os valores de entrada associados a cada ação e como mapear essa ação com a página web que pretende testar. O mapeamento é igual para todos os tipos de ação que se podem adicionar, sendo somente necessário preencher a informação de como o encontrar na página web. De modo a tornar este processo mais fácil e automático, foi criado uma técnica que permite ao utilizador seleccionar o elemento que mapeia a ação diretamente na página web, depois de pressionado o botão de seleção de elemento (botão verde da Figura 16) no TOM Editor.

Assim que esta opção se encontra ativa, o utilizador pode navegar com o rato por cima dos elementos da página web sob teste, e com a ajuda de uma caixa que se sobrepõem aos elementos HTML, clicar naquele que pretende mapear para a ação, como se pode verificar na Figura 17. Nesta figura vemos que o texto "*408 Results Found in 1ms*" aparece seleccionado pela caixa de seleção dos elementos.

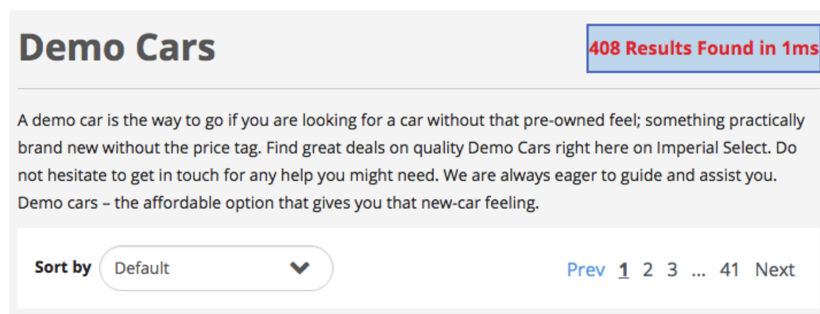


Figura 17.: Exemplo de seleção do texto na página web.

Depois de seleccionado o elemento que se pretende mapear, são realizadas na componente do "*content script*" algumas verificações de forma a obter o caminho mais correto e válido até ao elemento. Inicialmente é verificado o tipo do elemento HTML, caso o tipo seja um "a (*anchor*)", verifica-se se o atributo *href* contém um valor válido, isto é, um *url* para outra página. Se o tipo do elemento for "*input*" obtém-se o valor do atributo *name*. Posteriormente, se o tipo de elemento seleccionado não for nenhum dos anteriormente re-

feridos ou não se encontrou um valor válido nos atributos anteriores, verifica-se os outros atributos que compõem o elemento. Começa-se por verificar se o atributo *id* do elemento está definido. Caso não se encontre, analisa-se se o atributo *class* está com um valor válido para mapeamento, isto é, se o valor da classe é único na página web. Caso ainda não se tenha encontrado um mapeamento válido, é executado um método que obtém o caminho através de seletores CSS.

Definida a forma como se encontra o elemento, é criada uma mensagem com o atributo do elemento HTML seleccionado ou o seletor CSS, com o correspondente valor, e enviada para a componente *connection service* do TOM Editor os dados obtidos. Neste componente, depois de recebida a mensagem, identifica-se qual o formulário que solicitou o mapeamento através da seleção do elemento na página web e atribuem-se os valores definidos para cada variável. Na Figura 18, é possível visualizar a parte do mapeamento no formulário alterada de acordo com os dados recebidos da mensagem.

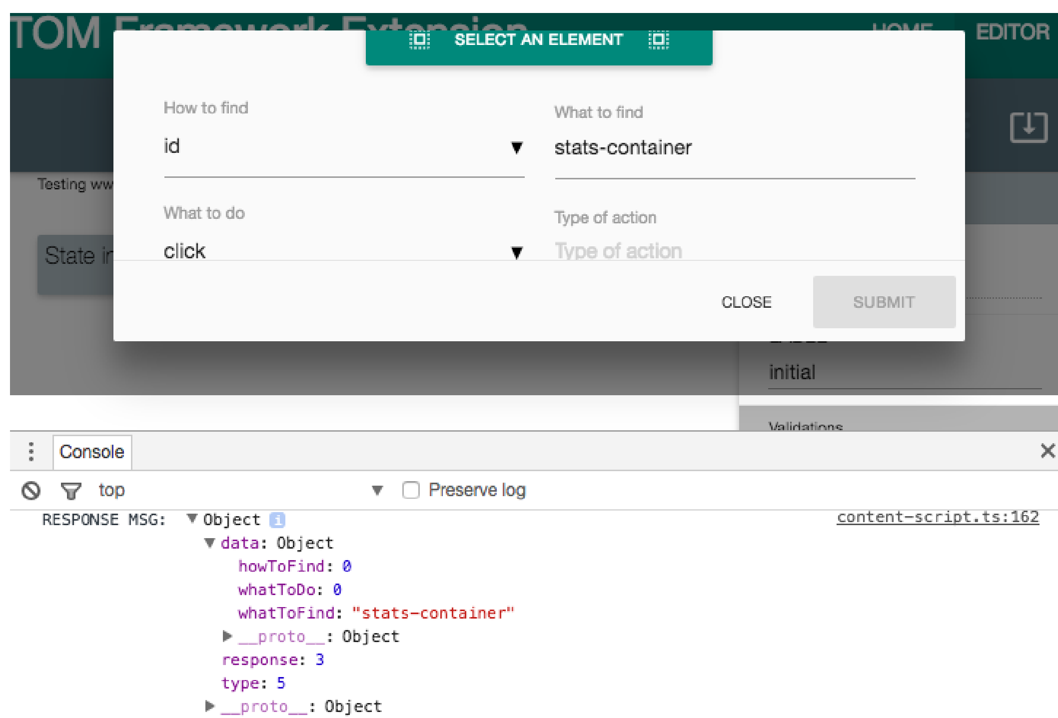


Figura 18.: Mapeamento de formulário alterado, após seleção direta de elemento na página web.

5.6 CAPTURA DA INTERAÇÃO DO UTILIZADOR

O modo de captura da informação de utilização que o utilizador realiza numa página web é conseguido com base num *script* que é injetado na camada do DOM da página que se

está a testar. O *script* contém dois *event listeners* que para determinadas características de utilização, como cliques em *links* ou preenchimento de formulários, constroem mensagens que são enviadas para o TOM Editor, onde é realizado o processamento da mensagem e adição dessa ação no editor. A Figura 19 mostra-nos onde é possível ativar o modo de captura no editor.

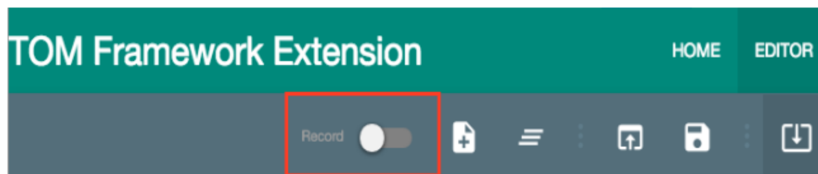


Figura 19.: Interface gráfica para a ativação do modo de captura de utilização.

Os *event listeners* utilizados para capturar a interação na DOM são o *click* e o *blur*. O *click* é responsável por determinar se o utilizador pretende ir para uma nova página, isto é, se efetuou um clique numa hiperligação que origine uma transição ou se carregou num botão de submissão de um formulário na página web, originando assim, respectivamente, uma transição ou um novo formulário. O evento *blur* acciona uma ação quando um elemento da DOM perde o foco, por exemplo, quando se acaba de preencher um campo do *input* e se salta para outra interação na página. A ação despoletada verifica se o elemento era do tipo *input* ou *textarea*, e em caso afirmativo obtém o valor dos dados lá inseridos pelo utilizador, para de seguida construir e enviar uma mensagem para a aplicação com esta interação de introdução de dados num formulário da página web.

Antes da informação ser enviada para a aplicação, e recorrendo à estratégia definida de seleção do elemento da DOM (explicada na secção anterior) é obtido o melhor caminho que mapeia esta nova ação com a GUI da página web.

Assim que a mensagem é recebida na aplicação é realizada a validação e o tratamento da mesma, atribuindo ao estado ativo na aplicação o resultado desta nova ação. Esta ação ou é do tipo transição, novo formulário ou introdução de dados em formulários. Até este momento, ainda não se encontrou uma forma de agilizar a automação do processo de validação com base na interação do utilizador na página, uma vez que numa validação é necessário definir se ela se realiza ao entrar ou sair de um estado e qual o tipo de teste que deve ser efetuado naquele elemento.

Normalmente, a captura da interação do utilizador na página realiza-se de forma iterativa, isto é, quando o utilizador realiza uma transição para uma página e depois a partir desta realiza outra transição para uma nova página, isto origina a criação de três estados e duas transições (A ->B ->C). Isto acontece porque no TOM Editor sempre que existe a

transição para um novo estado, este fica marcado como ativo. No entanto, é possível seleccionar no TOM Editor o estado ativo a cada momento, basta clicar no estado que se pretende, e as ações são criadas a partir deste.

5.7 EXPORTAÇÃO DE DADOS

Como os dados se encontram armazenadas na memória da sessão o utilizador pode a qualquer momento exportar essa informação para ficheiros, que são essenciais para a geração dos testes no TOM Generator. Na Figura 20, vemos os tipos de configuração de ficheiros que são possíveis guardar. Para originar estes ficheiros é necessário realizar a leitura da estrutura de dados atual e para cada necessidade é executado um método que escreve num ficheiro os dados da máquina de estados, mapeamento, valores e mutações, com base nas regras definidas inicialmente na Secção 3.1

Entretanto, se o utilizador pretender guardar o modelo do sistema completo, é exportado um ficheiro com toda a informação introduzida no TOM Editor. É mais tarde possível carregar esta informação para a aplicação, e continuar o desenvolvimento do modelo.

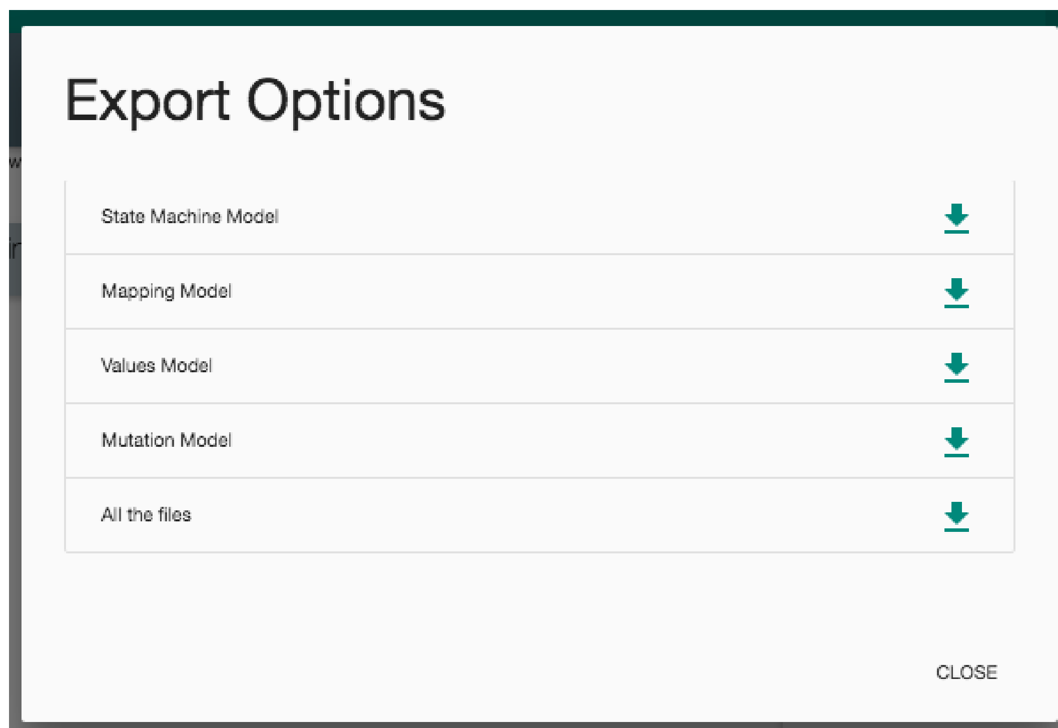


Figura 20.: Interface gráfica com os tipos de configurações exportáveis.

5.8 CONCLUSÕES

Neste capítulo foram apresentados as fases de implementação do TOM Editor, com alguma discussão dos problemas e soluções encontrados no decorrer do mesmo.

Na fase inicial de implementação foram sentidas algumas dificuldades no evolução da ferramenta, muito por causa da falta de experiência na utilização da tecnologia escolhida. Depois deste problema ultrapassado, a maior dificuldade sentida foi na ligação da ferramenta ao browser, isto é, na comunicação entre os conteúdos que estão ao mesmo nível que a página e a extensão que se encontra num painel da janela de inspeção. Depois de analisada profundamente a documentação sobre o desenvolvimento de extensões foi possível conseguir ligar todos os componentes e terminar a extensão. O conhecimento e experiência, ganhos nesta dissertação, fazem com que esta fase fosse das mais interessantes de todo o processo.

Na Figura 21, encontramos um resumo das principais funcionalidades da extensão, o TOM Editor. O modo editor para construir e editar o modelo do sistema, o modo de captura da interação do utilizador na aplicação web, por conseguinte a construção automática da máquina de estados, o modo de seleção de um elemento na página web, de modo a tornar o mapeamento mais fácil.

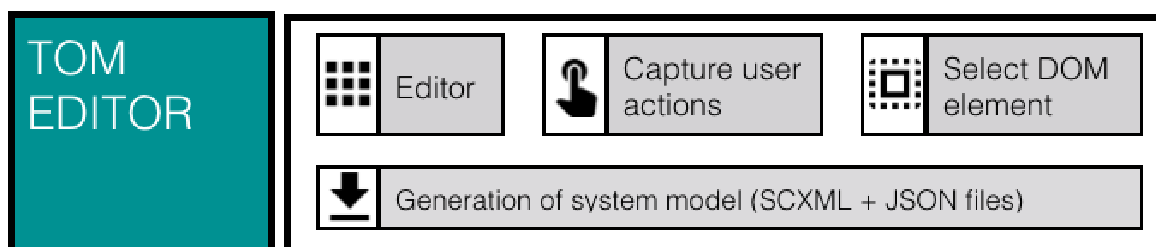


Figura 21.: Funcionalidades disponíveis a partir do TOM Editor.

6

APLICAÇÃO DA TOM FRAMEWORK

Neste capítulo pretende-se demonstrar a aplicação da *framework* a uma aplicação web. Com base nas funcionalidades existentes nas ferramentas que constituem a *framework*, pretende-se criar um modelo do sistema com recurso ao TOM Editor, utilizando-o no TOM Generator para gerar e executar casos de teste às aplicações, sendo no final analisados e discutidos os resultados obtidos nos diferentes tipos de configurações.

A aplicação web escolhida foi o OntoWorks (`http://ontoworks.epl.di.uminho.pt`), um *endpoint* genérico de SPARQL que permite a adição de ontologias e a execução de *queries* sobre estas com a possibilidade de guardar as queries SPARQL associadas à ontologia. Esta aplicação foi desenvolvida no decorrer de uma unidade curricular do Mestrado em Engenharia Informática e não foi sujeita a um processo de teste exaustivo. Como tal, vamos usar o algoritmo de todos os caminhos para obter o maior número de casos de testes possíveis. No final, para os diferentes casos de teste realizados, vamos tentar perceber se foi encontrado algum erro.

O OntoWorks permite o acesso, visualização e execução de *queries* a ontologias. É possível verificar quais a lista de ontologias disponíveis e ter acesso à listagem de queries associadas a cada ontologia. O utilizador tem a possibilidade de se registar e autenticar na plataforma. Depois de realizada a autenticação o utilizador pode carregar, editar e remover ontologias do sistema, assim como associar e remover *queries* das ontologias.

6.1 MODELO DO SISTEMA

Com recurso ao TOM Editor começou-se a construir o modelo do sistema que irá definir a aplicação web sobre teste. Para nos ajudar em parte deste processo, foi ativado o modo de captura de interação do utilizador com a página web. Deste modo, a construção dos estados com transições e formulários aconteceu de forma praticamente imediata e contínua. Na Figura 22 podemos ver uma parte do modelo construído com a captura da interação

ativa. No estado "home" existem duas transições e um formulário. As transições ocorrem para novos estados, "About us" e "Sign up". O formulário, de introdução de dados do login, dá acesso ao estado "root", depois de submetido.

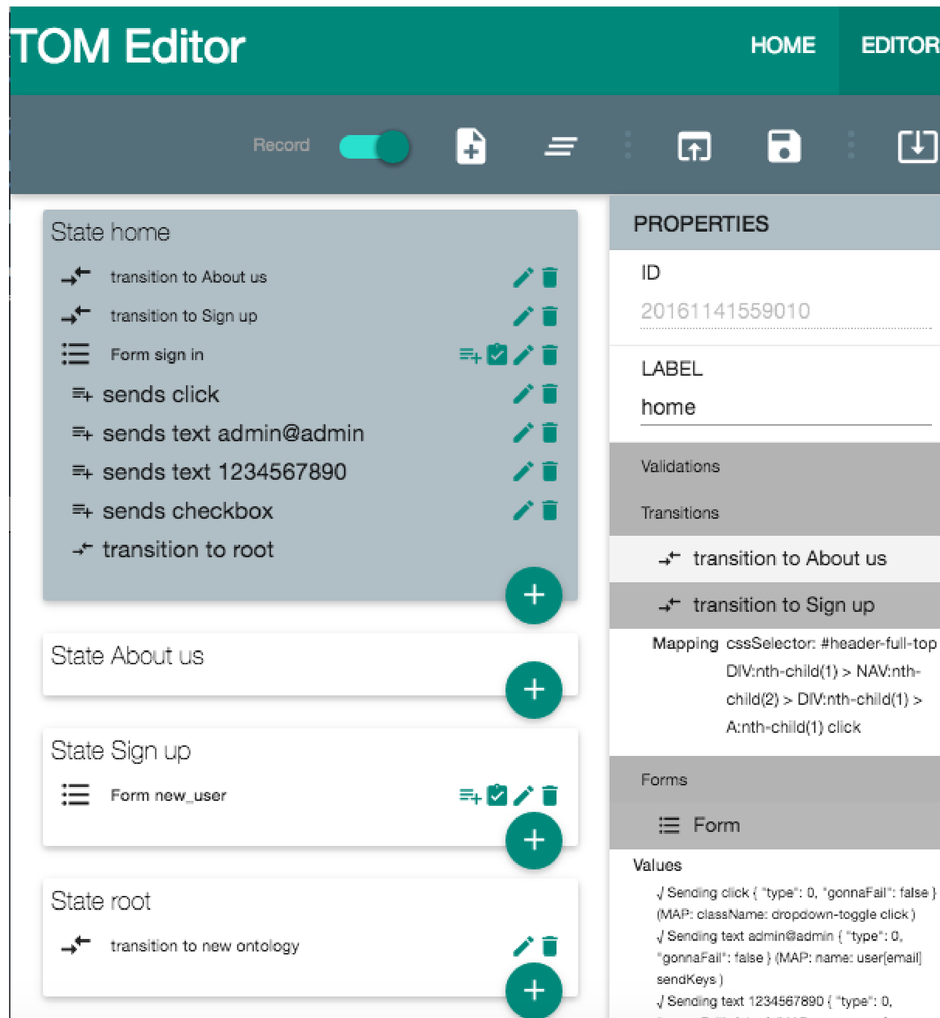


Figura 22.: Interface gráfica com parte do modelo do sistema construído.

A Figura 23 mostra-nos uma parte da barra de navegação da aplicação web, que corresponde em parte à navegação feita pelo utilizador na construção do modelo da figura anterior (22). A barra de navegação permite sempre acesso às outras páginas web ("Home", "Ontologies", "About us", "Sign up"). Optamos por só representar as transições a partir da página "Home", de forma a tornar o modelo do sistema menos complexo.

Terminada a fase de construção das transições e formulários do sistema, foram adicionadas aos estados, manualmente, as validações que permitem verificar e validar se ao entrar ou sair do estado tudo se encontra de acordo com o que se pretende testar. Para tal, recorremos aos formulários de criação das validações, onde podemos escolher se a mesma é

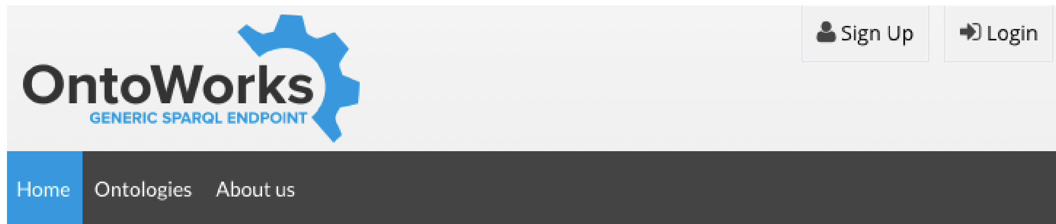


Figura 23.: Interface gráfica da barra de navegação do OntoWorks.

realizada a entrar ou sair do estado e o tipo de teste efetuado. Posteriormente, foi necessário definir o mapeamento da validação com a página web. Para nos ajudar neste processo utilizamos o modo de seleção direta do elemento na página web que queremos validar.

A Figura 24 apresenta um formulário de criação de uma validação. Pretende-se com esta validação verificar se o logo da aplicação web é apresentado no ecrã quando se entra no estado "home". Para facilitar o mapeamento foi utilizado a seleção do elemento na página web. Como se percebe pela figura, a caixa azul que se sobrepõem indica que é possível obter o mapeamento do logo.

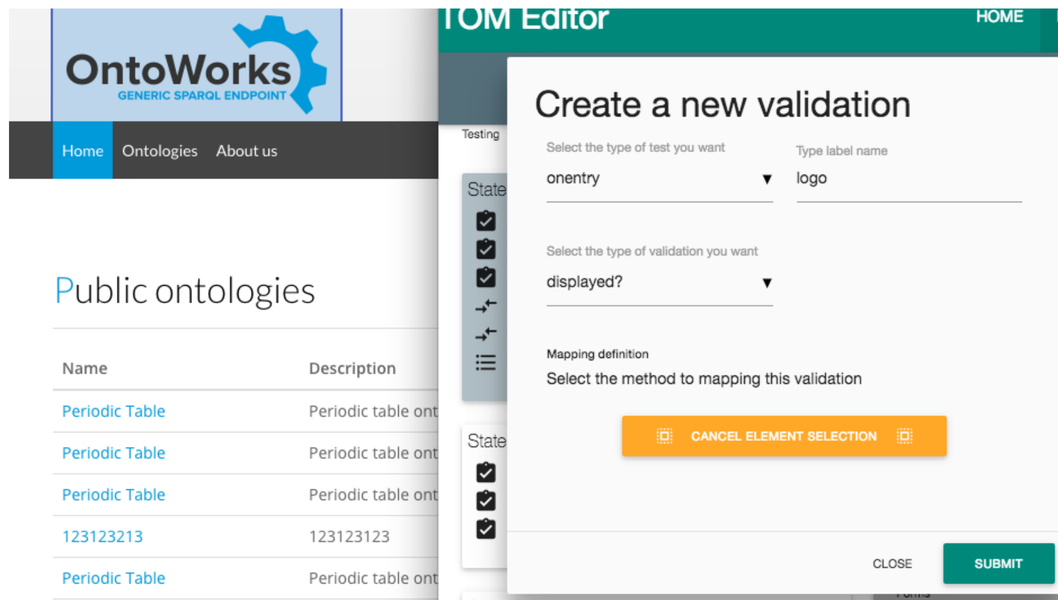


Figura 24.: Formulário de criação de uma validação e seleção de elemento na página web.

A Figura 25 mostra-nos algumas das validações que foram adicionadas ao estado "home". Neste estado foram criadas quatro validações, três que se realizam ao entrar no estado, sendo que estas acontecem antes do preenchimento do formulário, e uma que acontece depois do formulário ser preenchido. A primeira validação, identificada na figura pela caixa verde e explicada anteriormente, verifica se o logo é apresentado. A segunda validação,

apresentada na caixa roxa, verifica se na *navbar* da página web o elemento "Home" está com cor de fundo azul (*rgba(0, 153, 218, 1)*). A terceira validação, caixa amarela, verifica se o botão de *login* contém o texto "Login". Por último, a verificação na caixa vermelha, só ocorre depois de o formulário ser submetido, verifica se surgiu o alerta de autenticação com sucesso na página web.

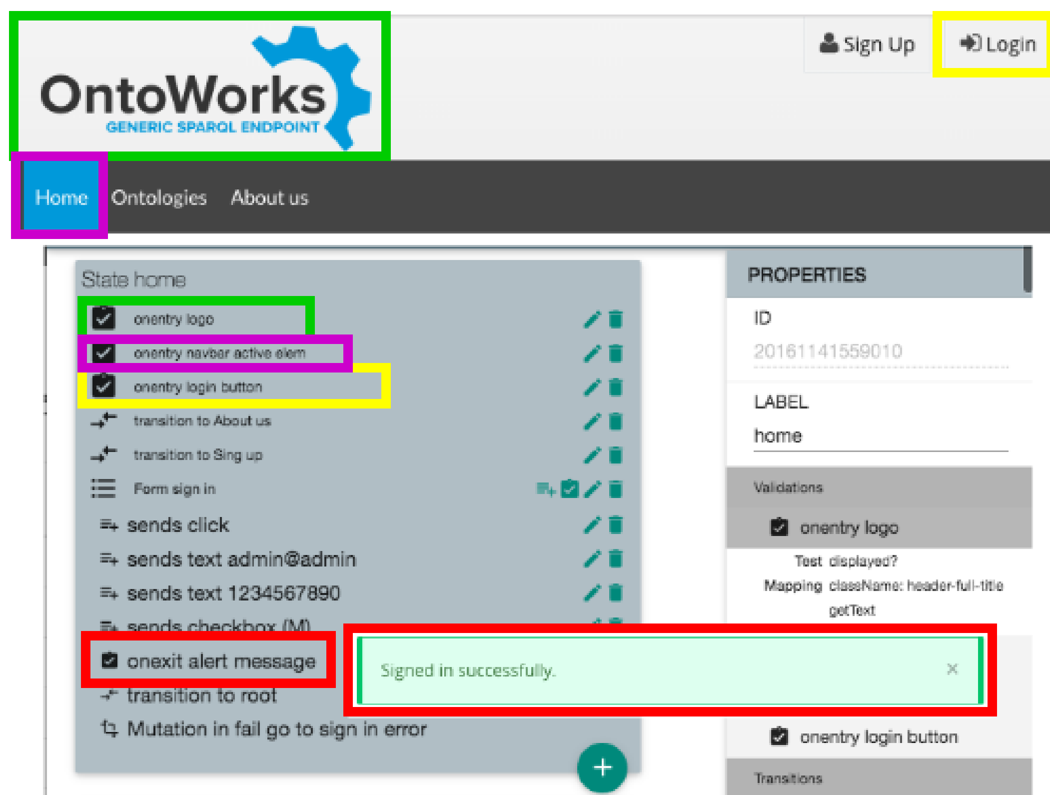


Figura 25.: Interface gráfica com as validações do estado "home".

Outro dos aspectos que é visualizado na figura anterior (Figura 25), é a existência de mutações no formulário. As mutações são, por norma, introduzidas manualmente depois de já se ter obtido as informações gerais de constituição do formulário. Neste caso, foi introduzida uma mutação de esquecimento de preenchimento do formulário (*lapse*), no campo *checkbox*. Na Figura 26 é apresentada, em parte, a janela de configuração do envio de valores para o formulário, com as definições da mutação anterior. Como vemos na figura, o campo *"Gonna fail this test?"* não se encontra seleccionado, logo a mutação definida não afetará o decorrer normal dos testes.

Ainda no formulário do estado "home", da Figura 25, encontra-se uma transição (*Mutation in error go to ...*) para um novo estado (*sign in error*). Esta transição foi definida para os casos em que a mutação escolhida interfere no decorrer normal do teste. A mutação definida na Figura 26 não interfere com o decorrer normal dos testes, no entanto no de-

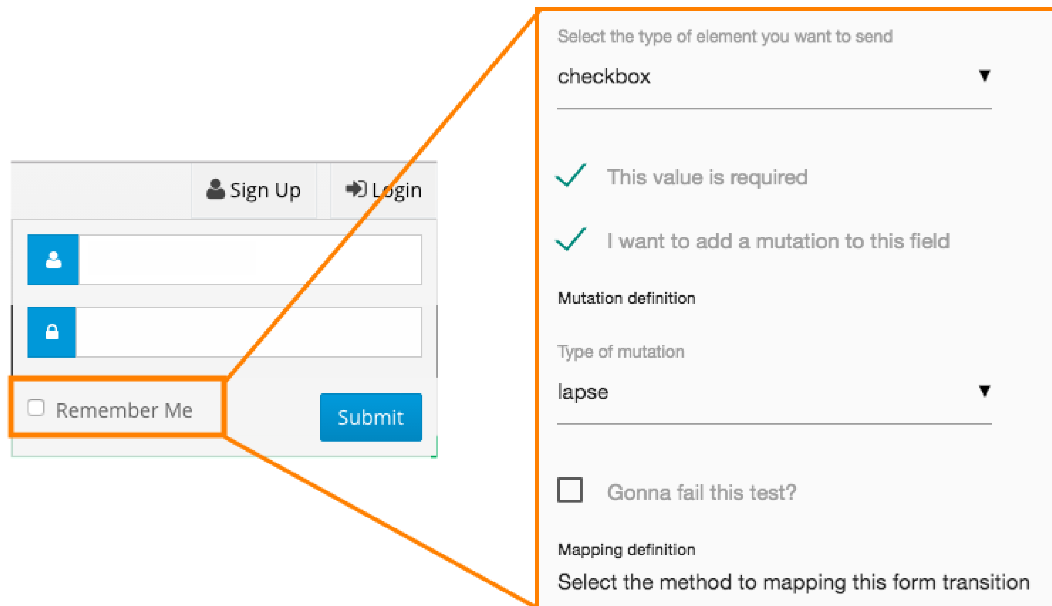


Figura 26.: Interface gráfica com as definições das mutações.

correr da geração automática de casos de teste podem ser introduzidas mutações de falha. Quando esta situação ocorre no preenchimento de dados do formulário de *login*, existe uma transição para o estado (*sign in error*). Neste estado, apresentado na Figura 27, foi adicionado uma validação que verifica a existência de um alerta de erro na página web.

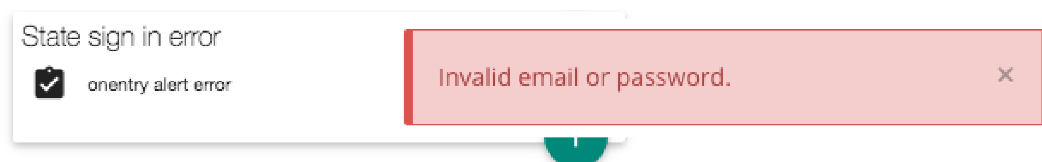


Figura 27.: Estado de erro (*sign in error*) com a validação existente.

O atual modelo do sistema pode ser descrito pela máquina de estados que se encontra na Figura 28. O modelo é composto por 15 estados, 5 dos quais representam estados de erro e estão apresentados na figura com um sombreado cinzento. Optou-se por não representar a navegação a partir destes estados por a mesma não ser relevante nesta avaliação. Existem 7 sub estados que correspondem ao preenchimento de formulários na aplicação web e 24 transições entre estados. Na figura não são demonstradas as 61 verificações, 57 ao entrar na página web e 4 ao sair, que foram adicionadas aos estados de forma a não tornar a interpretação da figura complicada. A validação "*displayed?*" foi a mais utilizada (27 vezes) nas verificações na página web.

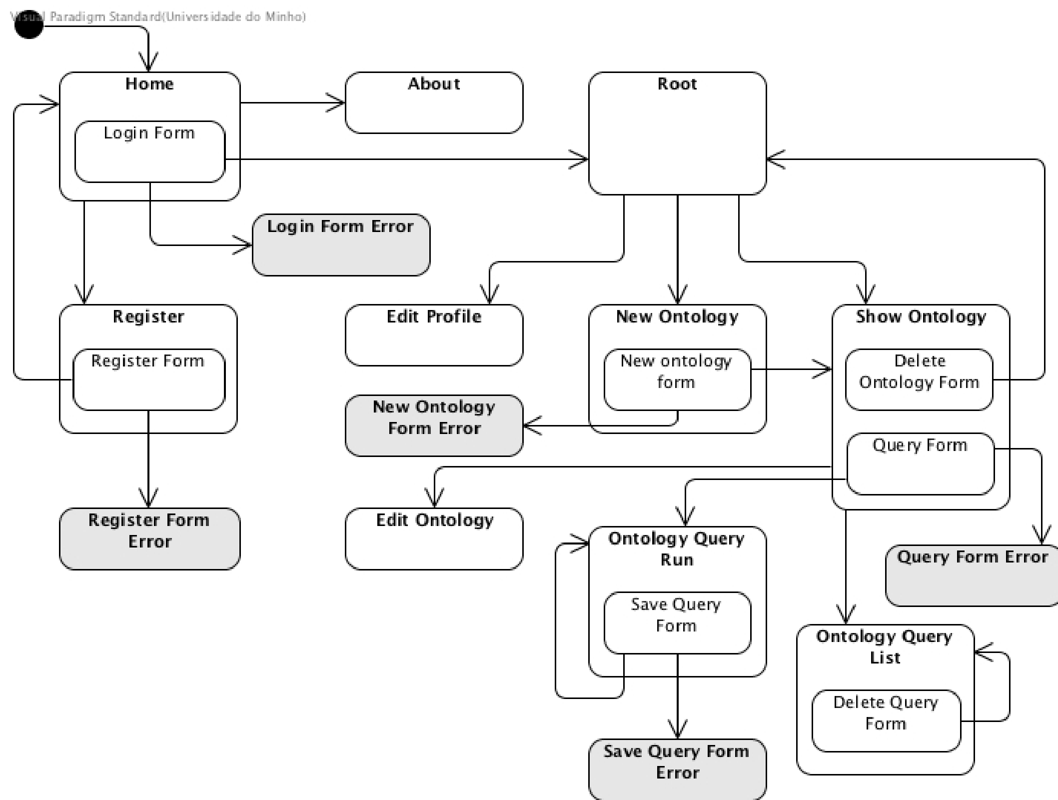


Figura 28.: Máquina de estados do OntoWorks.

A modelação do sistema do OntoWorks no TOM Editor demorou cerca de 5 horas, o que é uma redução bastante significativa comparada com o que era necessário anteriormente. Antes de se realizar a modelação do OntoWorks no TOM Editor foi feita a modelação deste sistema de forma manual, obtendo-se no final uma máquina de estados bastante similar à da Figura 28, que custou temporalmente 27 horas. O grande problema de se realizar a modelação manualmente é o custo existente em mapear cada elemento da máquina de estados com a página web. No atual modelo do sistema existem 102 mapeamentos, obtidos na grande maioria das vezes de forma automática.

Terminado o modelo do sistema foram exportados os quatro ficheiros que definem este sistema (Máquinas de estados, mapeamento, valores e mutações), que se encontram no Anexo A.1 desta dissertação.

6.2 CONFIGURAÇÕES NO TOM GENERATOR

As configurações de geração dos casos de teste são importantes para definir a forma como os mesmos são obtidos. Para este caso de estudo foram utilizadas as seguintes definições:

- O algoritmo de pesquisa de todos os caminhos (*AllPaths*) foi o seleccionado para percorrer o grafo obtido na leitura da máquina de estados, a partir do nodo inicial definido. De forma a garantir que o algoritmo percorre todos os vértices e arestas pelo menos uma vez, foram definidas as restrições seguintes: foi limitado a 1 o número de visitas a vértices e a 2 o número de visitas a arestas. Desta forma, garante-se uma cobertura total da aplicação nos casos de testes gerados.
- O Google Chrome foi o *browser* escolhido para a execução dos casos de teste, e em cada página web que se entre, no decorrer da execução, vai ser verificada a validade de todos os *links* e imagens. Em caso de erro é obtido um *screenshot* da página web.
- Foram seleccionadas todos os tipos de mutações para serem geradas com os casos de teste. Nesta situação, além do ficheiro com os casos de teste normais, vão ser gerados mais 10 ficheiros com casos de teste, um para cada tipo de mutação existente no TOM Generator, sendo que o último ficheiro contém as mutações definidas pelo utilizador.

A Listagem 6.1 apresenta as definições anteriormente mencionadas, mas no formato definido no TOM Generator.

```
/* PATH TO CONFIGURATIONS FILES */
private static final String PACKAGE_NAME = "ontoworks";
private static final String FILE_MAP="/Users/miguelpinto/Desktop/editor/mapping.json";
private static final String FILE_VALUES = "/Users/miguelpinto/Desktop/editor/values.
    json";
private static final String FILE_MUTATIONS = "/Users/miguelpinto/Desktop/editor/
    mutation.json";
private static final String FILE_MODEL = "/Users/miguelpinto/Desktop/editor/
    state_machine.xml";
private static final String URL = "http://ontoworks.epl.di.uminho.pt";

/* NUMBER OF GENERATED FILES TESTS & OUTPUT FOLDERS */
private static final int TEST_NUMBER = 10;
private static final String FOLDER_TESTS = "/Users/miguelpinto/Desktop/workspace/
    tom_generator/src/test/java/generated_tests/ontoworks/";

/* ALGORITHMS DEFINITIONS */
private static final string ALGORITHM = "ALLPATHS";
private static final String INITIAL_NODE = "20161141559010";
private static final int MAX_VERTEX_VISIT = 1;
private static final int MAX_EDGE_VISIT = 2;

/* SELENIUM DEFINITIONS */
private static final int BROWSER = 1;
private static final int SCREENSHOT_ON_ERROR = 1;
private static final int BROKEN_LINKS = 1;
private static final int BROKEN_IMAGES = 1;
```

Listing 6.1: Exemplo de configuração do TOM Generator para a aplicação OntoWorks.

6.3 GERAÇÃO E EXECUÇÃO DOS CASOS DE TESTE

A leitura e geração dos casos de teste processa-se de forma bastante rápida no TOM Generator. Inicialmente, a partir da leitura e interpretação do ficheiro com a máquina de estados é criado um grafo com a informação do sistema. De seguida, é aplicado ao grafo o algoritmo de pesquisa de todos os caminhos, e são obtidos 12 caminhos abstratos (independentes da tecnologia), cada um, dividido em várias etapas. O primeiro caminho descrito pelo TOM Generator, que vai ao encontro, em parte, do exemplo explorado até ao momento (preenchimento do formulário de *login*), é o seguinte:

1. Entra na *homepage* e verifica se o logo da aplicação está visível.
2. Verifica se a cor do elemento ativo na navbar.
3. Verifica se o botão contém a informação pretendida.
4. Início do preenchimento do formulário de *login*:
   a) Clique para abrir o formulário de *login*.
   b) Preenche o campo *email*.
   c) Preenche o campo *password*.
   d) Clica no elemento da *checkbox*.
   e) Clica no botão de submissão do formulário de *login*.
5. Entra na página *root* e verifica o conteúdo do alerta que aparece na página.
6. Verifica se é visível na página um alerta do *login*.
7. Verifica se o nome da seção contém o texto pretendido.
8. Verifica se o nome de outra seção tem exatamente o texto definido no teste.
9. Verifica se a *label* com o nome do utilizador é igual ao pretendido.
10. Realiza uma transição para a página de carregar uma nova ontologia (*"New Ontology"*).
11. Verifica se o elemento *"New Ontology"* da navbar tem o atributo select como ativo.
12. Verifica se o campo do formulário para colocar o caminho de uma ontologia se encontra ativo.
13. Verifica se a seção do formulário está visível.
14. Início do preenchimento do formulário para carregar uma nova ontologia:
    a) Preenche o campo com o nome da ontologia.
    b) Preenche o campo com a descrição da ontologia.

c) Clica no elemento da *checkbox* para tornar a ontologia pública.

d) Clica no campo e selecciona o ficheiro com a ontologia.

e) Clica no botão de submissão do formulário com os dados da ontologia.

15. Entra na página com informações da ontologia carregada e verifica se o alerta que surgiu tem a cor pretendida.
16. Verifica se a mensagem do alerta contém um determinado texto.
17. Verifica se a *label* com o nome do utilizador contém um determinado valor.

Depois de se ter obtido todos os caminhos abstratos possíveis é realizada automaticamente a transformação destes caminhos em casos de testes executáveis. Os casos de teste tem um formato próprio em *TestNG*, com especificações em *Selenium*. A Listagem 6.2 apresenta um pedaço do código gerado para o preenchimento do formulário de *login*, relativo à parte 4 da lista anterior.

```
@Test(invocationCount = 1, groups = "2")
public void path02_test004()throws IOException, InterruptedException {
    int gonna_fail = 0;
    // Form sign in
    try {
        driver.findElement(By.className("dropdown-toggle")).click();
        driver.findElement(By.name("user[email]")).sendKeys("admin@admin");
        String s20161141559033 = driver.findElement(By.name("user[email]")).
            getAttribute("value");
        driver.findElement(By.name("user[password]")).sendKeys("1234567890");
        String s20161141559034 = driver.findElement(By.name("user[password]")).
            getAttribute("value");
        driver.findElement(By.className("translation_missing")).click();
        Reporter.log("[Pass]:␣Form␣Filled␣sign␣in␣<br>");
        gonna_fail = 0;

    } catch (WebDriverException _x) {
        Reporter.log("␣[FAIL]:␣Error␣in␣Form␣Fill␣20161141559037␣" + _x.getMessage());
        fail("[FAIL]:␣Error␣in␣Form␣Fill␣20161141559037␣" + _x.getMessage());
    }

    driver.findElement(By.cssSelector("#header-full-top␣>␣DIV:nth-child(1)␣>␣NAV:nth-
        child(2)␣>␣DIV:nth-child(2)␣>␣DIV:nth-child(2)␣>␣FORM:nth-child(1)␣>␣
        BUTTON:nth-child(6)")).submit();
    Reporter.log("[Pass]:␣Click␣Submit␣Button␣␣<br>");
    Reporter.log("[Pass]:␣Form␣Completed␣␣ID:sign␣in␣<br>");
    my_wait();
}
```

Listing 6.2: Código gerado para o preenchimento do formulário de *Login*.

Como referido acima, foram gerados 11 ficheiros, o primeiro sem a introdução de mutações, o último com as mutações definidas por nós, no TOM Editor, e os restantes com mutações aleatórias em determinados campos para cada tipo de mutação existente na TOM Framework. Na Figura 29 é apresentado o pacote com os ficheiros de teste obtidos. No total existem 3003 casos de teste, 273 casos por cada ficheiro de teste, que podem ser realizados à aplicação web.

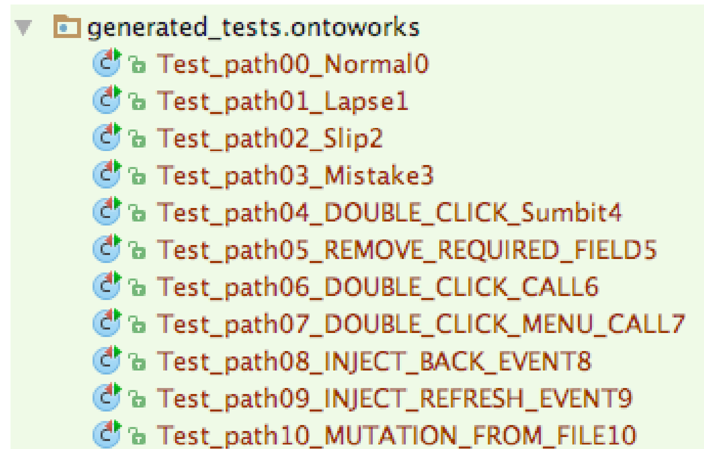


Figura 29.: Ficheiros de teste automaticamente gerados.

Finalmente, é possível executar no OntoWorks os casos de teste obtidos. Na próxima seção vamos analisar os resultados conseguidos.

6.4 RESULTADOS E DISCUSSÃO

Depois de executados os casos de teste, que se encontram na Figura 29, no OntoWorks foram encontrados 935 falhas em 3003 testes. Como é possível perceber por este número a plataforma foi largamente testada. Na Figura 30 encontra-se uma figura com o relatório do resultado final da execução dos casos de teste. Nesta figura é possível perceber que a duração total dos testes foi superior a 3 horas.

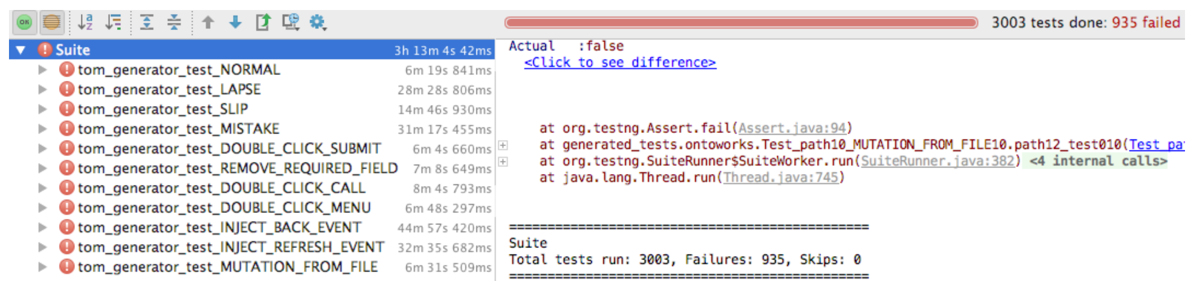


Figura 30.: Resultados da execução dos casos de teste ao OntoWorks.

Através da análise da figura anterior não é possível perceber em que tipos de teste é que existiram mais falhas. Assim foi criada a Tabela 9, que contém os resultados obtidos com o tempo de execução total de forma mais organizada e concisa.

Tipo de teste	Testes falhados	Tempo de execução
NORMAL	3	6m 19s
LAPSE	168	28m 28s
SLIP	67	14m 46s
MISTAKE	196	31m 17s
DOUBLE CLICK SUBMIT	3	6m 4s
REMOVE REQUIRED FIELD	23	7m 8s
DOUBLE CLICK CALL	15	8m 4s
DOUBLE CLICK MENU	6	6m 48s
INJECT BACK EVENT	261	44m 57s
INJECT REFRESH EVENT	182	32m 35s
MUTATIONS FROM FILE	11	6m 31s

Tabela 9.: Tabela com os resultados da execução dos casos de teste no OntoWorks.

Como é possível perceber pela tabela anterior (9), foram obtidos no total 935 falhas no decorrer da execução dos testes. Este valor à primeira vista não é um bom indicador e, como tal, fez-se uma análise mais aprofundada ao relatório gerado da Figura 30. Começou-se por analisar os resultados do ficheiro que contém os casos de teste normais.

Na Figura 31 são visualizados os 3 problemas encontrados, que ocorrem no mesmo caminho, e a explicação do problema do teste (*path12_test007*). Neste teste existe um problema na localização do elemento na página web, que não se encontra visível. Este erro acontece porque o *Selenium* está a tentar preencher um campo do formulário de *login* que está oculto na altura da execução.

Figura 31.: Problemas encontrados no caminho dos testes.

Analisando mais a fundo os elementos HTML da página web com o formulário de registo de um novo utilizador, apresentado na Figura 32, verifica-se que existem dois elementos com as mesmas definições em formulários diferentes. O elemento *input* contém dois atributos, o *id* e o *name*, que tem exatamente o mesmo valores nas variáveis dos dois formulários. O valor do atributo *id* deve ser único numa página web, conforme Hickson et al. (2014), daí que estejamos perante um erro de implementação da aplicação.

Figura 32.: Comparação dos elementos HTML dos formulários de registro e *login* da aplicação.

Os testes programados que se seguem (*path12_teste08 e path12_teste10*), representados na Figura 31, estão dependentes da submissão com sucesso do formulário de registo. Como aconteceu um erro que não estava previsto no caminho dos testes, isto é, o formulário não foi preenchido corretamente, os testes seguintes vão falhar nas suas validações. Esta falha vai também afetar todos os outros tipos de teste.

Os ficheiros com os tipos de mutações *LAPSE, SLIP, MISTAKE, INJECT BACK EVENT* e *INJECT REFRESH EVENT* são os que apresentam uma taxa mais alta de falhas na sua execução e no tempo de execução. De lembrar, que estes ficheiros são gerados automaticamente pela ferramenta, onde as mutações são introduzidas aleatoriamente em determina-

das partes do caminho.

Na Listagem 6.3 é apresentada uma parte do código gerado para uma mutação *Lapse*. Como vemos o campo com o valor do email foi retirado do preenchimento do formulário de *login*. Quando o formulário for submetido irá haver uma falha, por o campo *email* não estar preenchido. Neste caso, a ferramenta coloca automaticamente o valor da variável ”*gonna_fail*“ igual a 1, de forma a permitir que sejam realizadas as validações definidas no estado de erro (caso elas existam).

```
Reporter.log("␣Mutation␣Lapse␣<br>");
driver.findElement(By.className("dropdown-toggle")).click();
driver.findElement(By.name("user[password]")).sendKeys("1234567890");
s20161141559034 = driver.findElement(By.name("user[password]")).getAttribute("value");
Reporter.log("[Pass]:␣Form␣Filled␣sign␣in␣<br>");
gonna_fail = 1;
...
    if (gonna_fail == 1) {
    ...
    try {
        // Started validation for an error page
        assertTrue(driver.findElement(By.cssSelector("#sb-site␣>␣DIV:nth-child(1)␣>␣
            SECTION:nth-child(3)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child
            (1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)")).
            isDisplayed());
        Reporter.log("Element␣displayed?␣" + driver.findElement(By.cssSelector("#sb-
            site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-child(1)␣>␣
            DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣
            >␣DIV:nth-child(1)")).isDisplayed());
    }
    ...
    Reporter.log("␣Mutation␣Lapse␣killed␣<br>");
    return;
    ...
```

Listing 6.3: Exemplo de código gerado para a mutação *lapse*.

Como acontece esta mutação no formulário e uma vez que fica impossível continuar a seguir o caminho determinado porque o *login* do utilizador falhou, todos os testes seguintes daquele caminho vão falhar. Acontece que, a geração do caminho deveria terminar depois da existência de uma mutação de falha, o que não está a acontecer, obtendo-se assim dezenas de falhas nos testes. Visto que este tipo de falha acontece em grande parte das mutações automáticas, originando um número bastante grande de falhas nos testes, conclui-se que não é possível retirar conclusões sobre a relação da execução dos casos de teste com mutações na aplicação web.

Os restantes tipos de mutação, *REMOVE REQUIRED FIELD, DOUBLE CLICK CALL, DOUBLE CLICK MENU* e *MUTATIONS FROM FILE* apresentam um número reduzido de falhas, que são provocados pelo motivo anteriormente falado.

A mutação do tipo *INJECT BACK EVENT* introduz no final da primeira etapa de cada caminho um evento para retroceder na página, originado que todos os testes seguintes falhem. Já a mutação do tipo *INJECT REFRESH EVENT* é introduzida depois de se clicar no botão de submissão dos formulários.

6.5 CONCLUSÕES

Neste capítulo foi demonstrada a aplicação da TOM framework à aplicação Web OntoWorks. Embora não descrito neste capítulo, a *framework* foi também aplicada a uma outra aplicação, a aplicação CustoJusto (`http://www.custojusto.pt/`). Esta é uma aplicação mais madura, utilizada diariamente por centenas de pessoas na compra e venda de uma grande variedades de artigos (casas, carros, telemóveis, computadores). Os resultados obtidos estão em linha com os descritos para a aplicação OntoWorks.

Em conclusão, é possível concluir que a TOM Framework tem potencial para auxiliar na detecção de erros em aplicações Web. No entanto, é ainda necessário algum trabalho de melhoramento dos casos de teste com mutações gerados automaticamente.

7

CONCLUSÕES E TRABALHO FUTURO

Neste capítulo é realizada uma análise ao trabalho desenvolvido no decorrer desta dissertação. Inclui, no final, algumas sugestões de melhorias e novos desenvolvimentos que possam futuramente integrar a TOM Framework.

O TOM Generator é uma ferramenta que permite a geração de múltiplos casos de teste a partir do modelo do sistema que queremos testar. Este processo é rápido, fácil, automático e barato. No entanto, o maior custo deste processo acaba por ser a construção do modelo do sistema, que acontece de forma manual e sem controlo. Desta forma, podem, eventualmente, surgir erros na definição do comportamento na GUI e/ou no mapeamento entre o modelo e a aplicação, que afetam os casos de teste gerados.

Os principais objetivos desta dissertação consistiam em tornar o processo de criação do modelo do sistema mais fácil, rápido e estruturalmente sem erros. De forma a obter estas características foi desenvolvido o TOM Editor, uma extensão para o Google Chrome, que permite a criação do modelo de forma mais interativa e automatizada, isto é, com recurso à captura da interação do utilizador e seleção direta dos elementos HTML na página sobre teste.

A arquitectura da ferramenta adopta uma solução modular de forma a torná-la mais flexível e adaptável a diferentes contextos. A parceria desenvolvida com o IRIT permitiu comprovar que a modularidade existente no TOM Generator é uma mais valia, conseguindo-se uma reutilização bastante substancial. Foi um trabalho positivo, onde a partir de modelos de tarefas, transformados em máquinas de estados, se conseguiu gerar cenários. Alguns dos cenários obtidos foram executados pelo IRIT, tendo-se vindo a comprovar a possibilidade de detecção de desvios da aplicação sob teste em relação ao modelo.

Os resultados obtidos na aplicação da *framework* ao caso de estudo demonstra que a aplicação consegue detetar problemas na implementação das aplicações, nomeadamente no erro existente na aplicação OntoWorks. Os múltiplos casos de teste gerados, e apesar

dos problemas criados pela introdução automática de mutações em alguns casos de teste, fazem com que seja possível testar largamente as aplicações web a um baixo custo. O TOM Editor veio contribuir, ainda mais, para esta diminuição de custos, visto que os modelos do sistema podem agora ser construídos de forma mais rápida e estruturalmente correta. Com isto, concluímos que os testes baseados em modelos asseguram, no final, uma melhoria da qualidade do sistema desenvolvido.

Em suma, os resultados alcançados no desenvolvimento desta dissertação foram:

- Construção de uma extensão para o Google Chrome (TOM Editor).
- Criação do modelo do sistema através dos formulários disponíveis no TOM Editor. É possível criar estados e adicionar a estes, formulários com informações de preenchimento, transições e validações.
- Estabelecimento de comunicação entre a página web sobre teste e o TOM Editor, com o objetivo de se capturar automaticamente a interação do utilizador e permitir a seleção direta dos elementos HTML nessa página.
- Extração dos ficheiros que constituem o modelo do sistema pelo TOM Editor, para posterior utilização no TOM Generator.
- Adaptação do TOM Generator à geração de cenários na parceria com o IRIT, através do desenvolvimento de um novo módulo.
- Contribuição na publicação de um artigo científico em parceria com o IRIT, existindo ainda um outro submetido para revisão e outro em fase final de preparação.
- Manipulação das configurações dos algoritmos na geração de casos de teste, no caso de estudo proposto.
- Execução dos casos de teste gerados a uma aplicação web escolhida. Posteriormente fez-se uma análise e discussão dos resultados obtidos.
- Criação de vídeos a demonstrar as funcionalidades principais do TOM Editor.

7.1 TRABALHO FUTURO

Foram identificadas algumas limitações e possíveis novas funcionalidades à medida que se utilizavam as ferramentas da TOM Framework na aplicação aos casos de estudo. Desta forma, são propostas para futuras iterações da *framework*, as seguintes características:

- Desenvolvimento de uma ferramenta para a geração de relatórios da execução dos casos de teste, onde se obtenham relatórios mais descritivos e simples de interpretar.

- Melhoria na geração dos casos de teste com mutações, para restringir os casos de teste que são executados quando se está perante uma mutação de erro.
- Correção na geração de formulários com mutações, nomeadamente quando a mutação é inserida num atributo de preenchimento opcional no formulário, sendo neste caso possível a continuar a executar os testes. A alteração pode acontecer na variável que controla se depois da submissão de um formulário a execução dos testes vai falhar.
- Ligação entre a TOM Framework e o trabalho de Silva and Campos (2013, 2014). Desenvolvimento de funcionalidades que permitam calcular a máquina de estados de uma interface web, de forma automática, com informação do mapeamento.
- Melhorias no sistema de captura da interação do utilizador nas páginas web, nomeadamente na utilização da barra de navegação quando esta é composta por múltiplas opções.
- Possibilidade de se integrar novas validações e eventos assíncronos.
- Implementação de novas características na geração de cenários do IRIT, com a introdução de mutações neste processo.

BIBLIOGRAFIA

Ana Barbosa, Ana C. R. Paiva, and José Creissac Campos. Test case generation from mutated task models. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems - EICS '11*, page 175, New York, New York, USA, 2011. ACM Press. ISBN 9781450306706. doi: 10.1145/1996461.1996516. URL http://portal.acm.org/citation.cfm?doid=1996461.1996516.

F. Belli. Finite state testing and analysis of graphical user interfaces. In *Proceedings 12th International Symposium on Software Reliability Engineering*, pages 34–43, 2001. ISBN 0-7695-1306-9. doi: 10.1109/ISSRE.2001.989456. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=989456.

Silvia Berti, Francesco Correani, and Giulio Mori. TERESA: a transformation-based environment for designing and developing multi-device interfaces. *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, 2004. ISSN 1-58113-703-6. doi: 10.1145/985921.985939. URL http://dl.acm.org/citation.cfm?id=985939.

José Creissac Campos, Camille Fayollas, Celia Martinie, David Navarre, Philippe Palanque, and Miguel Pinto. Systematic automation of scenario-based testing of user interfaces. *EICS 2016 - 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 138–148, 2016. doi: 10.1145/2933242.2948735. URL http://dx.doi.org/10.1145/2933242.2948735.

Paulo Jesus Cruz and José Creissac Campos. Ambiente de geração, mutação e execução de casos de teste para aplicações web. In Luís Magalhães and Beatriz Santos, editors, *Atas da Conferência Interação 2013*, pages 45–52. Universidade de Trás-os-Montes e Alto Douro, 2013.

Sanjeev Dhawan, Nirmal Kumar, and Shiva Saini. Model based testing considering steps, levels, tools & standards of software quality. *Journal of Global Research in Computer Science*, 2(4):44–54, 2011.

A Dix, J Finlay, G D Abowd, and R Beale. *Human-Computer Interaction*. Number January. 2004. ISBN 0130461091. doi: 10.1207/S15327051HCI16234.

K A Ericsson and H A Simon. *Protocol analysis: Verbal reports as data*, volume 23. The MIT Press, 1993. ISBN 0262550237. doi: 10.2307/3151491.

Martin Helander, Thomas K. Landauer, and Prasad V. Prabhu. *Handbook of human-computer interaction*. Elsevier, 1997. ISBN 9780444818621.

Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Navara, Edward O'Connor, and Silvia Pfeiffer. HTML5: A vocabulary and associated apis for html and xhtml, 2014. URL https://www.w3.org/TR/html5/dom.html#the-id-attribute.

IEEE. IEEE Standard Glossary of Software Engineering Terminology. *Office*, 121990(1):1, 1990. ISSN 0-7381-0391-8. doi: 10.1109/IEEESTD.1990.101064. URL http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=159342.

International Organization for Standardization. ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) - part 11: guidance on usability. *International Organization for Standardization*, 1998(2):28, 1998. ISSN 13594184. doi: 10.1038/sj.mp.4001776. URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=16883.

Jonathan Jacky, Margus Veanes, Colin Campbell, and Washington Wolfram Schulte. *Model-Based Software Testing and Analysis with C#*. 2008. ISBN 9780521886550.

Célia Martinie, David Navarre, Philippe Palanque, and Camille Fayollas. A generic tool-supported framework for coupling task models and interactive applications. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '15, pages 244–253, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3646-8. doi: 10.1145/2774225.2774845. URL http://doi.acm.org/10.1145/2774225.2774845.

Mikko Aleksi Makinen. *Model based approach to software testing*. Msc. thesis, Helsinki University of Technology, 2007. URL http://lib.tkk.fi/Dipl/2007/urn009573.pdf.

Alan Moore and David Redmond-Pyle. *Graphical User Interface Design and Evaluation: A Practical Process*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1995. ISBN 013315193X.

Rodrigo MLM Moreira, Ana C. R. Paiva, and Atif Memon. A pattern-based approach for gui modeling and testing. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 288–297. IEEE, 2013.

Glenford Myers. *The Art of Software Testing, Second edition*, volume 15. Wiley, 2004. ISBN 0-471-46912-2. doi: 10.1002/stvr.322. URL http://www.noqualityinside.com/nqi/nqifiles/TheArtofSoftwareTesting-SecondEdition.pdf.

Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: An innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1): 65–105, 2014. ISSN 15737535. doi: 10.1007/s10515-013-0128-9.

Jakob Nielsen and Rolf Molich. Heuristic Evaluation of user interfaces. *CHI '90 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 249–256, 1990. ISSN 1942-597X. doi: 10.1145/97243.97281.

Donald A. Norman. The Psychology of Everyday Things. *The Psychology of Everyday Things*, pages 1–104, 1988. ISSN 00029556. doi: 10.2307/1423268.

Jeff Offutt. Quality attributes of Web software applications. *IEEE Software*, 19(2):25–32, 2002. ISSN 07407459. doi: 10.1109/52.991329.

Ana C. R. Paiva. *Automated specification-based testing of graphical user interfaces*. PhD thesis, Universidade do Porto, 2006. URL http://sigarra.up.pt/feup/pt/publs_pesquisa.FormView?P_ID=23628.

Ana C. R. Paiva, Nikolai Tillmann, João CP Faria, and Raul FAM Vidal. Modeling and testing hierarchical guis. In *Proceedings of the 12th International Workshop on Abstract State Machines*, 2005.

F Paterno. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 2000. ISBN 1852331550. doi: 10.1057/palgrave.ejis.3000385. URL http://www.amazon.de/dp/1852331550.

Peter G. Polson, Clayton Lewis, John Rieman, and Cathleen Wharton. Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies*, 36:741–773, 1992. ISSN 00207373. doi: 10.1016/0020-7373(92)90039-N.

Eric S. Raymond and Rob W. Landley. The art of unix usability, 2004. URL http://www.catb.org/~esr/writings/taouu/html/index.html.

E.S. Raymond. The art of Unix programming. *System*, page 560, 2003. doi: 10.1002/chin.200327184. URL http://portal.acm.org/citation.cfm?id=829549.

James Reason. *Human Error*. Cambridge University Press, 1990. ISBN 9780521314190.

Jeremy Reimer. A history of the gui, 2005. URL http://arstechnica.com/features/2005/05/gui/.

Raphael Julien Rodrigues. *Testes Baseados em Modelos*. Msc. thesis, Universidade do Minho, 2015.

C. E. Silva and José Creissac Campos. Combining static and dynamic analysis for the reverse engineering of web applications. In P. Forbrig, P. Dewan, M. Harrison, K. Luyten, C. Santoro, and S.D.J. Barbosa, editors, *Proceedings of the 5th ACM SIGCHI Symposium on*

Engineering Interactive Computing Systems (EICS 2013), pages 107–112. ACM, 2013. URL http://doi.acm.org/10.1145/2494603.2480324.

C. E. Silva and José Creissac Campos. Characterizing the control logic of web applications' user interfaces. In *Computational Science and Its Applications - ICCSA 2014*, volume 8584 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2014. doi: 10.1007/978-3-319-09153-2_20.

José L. Silva, José Creissac Campos, and Ana C. R. Paiva. Model-based User Interface Testing With Spec Explorer and ConcurTaskTrees. *Electronic Notes in Theoretical Computer Science*, 208(C):77–93, 2008. ISSN 15710661. doi: 10.1016/j.entcs.2008.03.108.

Mark Utting and Bruno Legeard. Practical model-based testing: a tools approach. *Book*, page 433, 2007.

Maarten W van Someren, Yvonne F Barnard, and Jacobijn AC Sandberg. *The think aloud method: A practical guide to modelling cognitive processes*. Academic Press, London, 1994. ISBN 0127142703. doi: 10.1016/0306-4573(95)90031-4.

Marco Winckler and Marcelo Pimenta. Avaliação de Usabilidade de sites Web. *Nedel, Luciana (Org.) X Escola de Informática da SBC-Sul (ERI2002)*, pages 85–137, 2002. URL http://www.irit.fr/~Marco.Winckler/2002-winckler-pimenta-ERI-2002-cap3.pdf.

Dianxiang Xu. A tool for automated test code generation from high-level petri nets. In *Proceedings of the 32Nd International Conference on Applications and Theory of Petri Nets*, PETRI NETS'11, pages 308–317, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21833-0. URL http://dl.acm.org/citation.cfm?id=2022192.2022212.

MODELOS DO SISTEMA

A.1 ONTOWORKS

A.1.1 *Máquina de estados*

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" initialstate="20161141559010">
  <state id="20161141559010">
    <onentry id="2017031127001" type="displayed?" />
    <onentry id="2017031127002" type="css" />
    <onentry id="2017031127003" type="contains" />
    <transition id="20161141559012" target="20161141559011" />
    <transition id="20161141559017" target="20161141559016" />
    <state id="sign␣in" type="form">
      <send label="s20161141559032" type="required" element="checkbox" />
      <send label="s20161141559033" type="required" />
      <send label="s20161141559034" type="required" />
      <send label="s20161141559035" type="required" element="checkbox" />
      <transition type="form" label="20161141559037">
        <submit target="20161141559038" />
        <error target="20161121851003" />
      </transition>
      <onexit id="2016114208001" type="contains" />
    </state>
  </state>
  <state id="20161141559011">
    <onentry id="20161141559013" type="default" />
    <onentry id="20161141559014" type="displayed?" />
    <onentry id="20161141559015" type="url" />
  </state>
  <state id="20161141559016">
    <onentry id="20161141559026" type="displayed?" />
    <onentry id="20161141559027" type="css" />
    <state id="new_user" type="form">
      <send label="s20161141559019" type="required" />
      <send label="s20161141559020" type="required" />
      <send label="s20161141559021" type="required" />
      <send label="s20161141559022" type="required" />
      <transition type="form" label="20161141559024">
        <submit target="20161141559010" />
```

```
        <error target="20161121851006" />
      </transition>
      <onexit id="20161141559028" type="contains" />
    </state>
  </state>
  <state id="20161141559038">
    <onentry id="2016114208002" type="displayed?" />
    <onentry id="2016114208003" type="contains" />
    <onentry id="2016114208004" type="default" />
    <onentry id="2016114208007" type="default" />
    <transition id="2016114208009" target="2016114208008" />
    <transition id="20161121029002" target="20161121029001" />
    <transition id="20161121029018" target="20161121029017" />
  </state>
  <state id="2016114208008">
    <onentry id="2016114208019" type="attribute" />
    <onentry id="2016114208020" type="enabled?" />
    <onentry id="2016114208021" type="displayed?" />
    <state id="new_ontology" type="form">
      <send label="s2016114208010" type="required" />
      <send label="s2016114208011" type="required" />
      <send label="s2016114208012" type="required" element="checkbox" />
      <send label="s2016114208014" type="required" />
      <transition type="form" label="2016114208016">
        <submit target="2016114208017" />
        <error target="20161131049001" />
      </transition>
    </state>
  </state>
  <state id="2016114208017">
    <onentry id="2016114208018" type="css" />
    <onentry id="20161121029022" type="contains" />
    <onentry id="20161121029023" type="default" />
    <onexit id="20161121029024" type="enabled?" />
  </state>
  <state id="20161121029001">
    <onentry id="20161121029025" type="displayed?" />
    <onentry id="20161121029026" type="enabled?" />
    <onentry id="20161121029027" type="displayed?" />
    <onentry id="20161121029028" type="default" />
    <onentry id="20161121029029" type="displayed?" />
    <onentry id="20161121029030" type="contains" />
    <onentry id="20161121029031" type="displayed?" />
    <transition id="20161121029016" target="20161121029015" />
    <transition id="20161121029019" target="20161121029011" />
    <transition id="20161121029021" target="20161121029020" />
    <state id="submit_query" type="form">
      <send label="s20161121029003" type="required" element="selectbox" />
      <send label="s20161121029004" type="required" element="checkbox" />
      <send label="s20161121029005" type="required" element="checkbox" />
      <send label="s20161121029006" type="required" element="checkbox" />
      <send label="s20161121029007" type="required" />
      <send label="s20161121029034" type="required" element="checkbox" />
      <send label="s20161131049005" type="required" element="checkbox" />
      <transition type="form" label="20161121029009">
```

```
      <submit target="20161121029010" />
      <error target="20161131049004" />
    </transition>
    <onexit id="2017031127004" type="enabled?" />
  </state>
  <state id="alert␣delete␣ontology" type="form">
    <send label="s20161121425024" type="required" element="checkbox" />
    <transition type="alert" label="20161121425023">
      <submit target="20161141559038" />
    </transition>
    <onexit id="20161121425027" type="displayed?" />
  </state>
</state>
<state id="20161121029010">
  <onentry id="20161121029038" type="displayed?" />
  <onentry id="20161121029039" type="not_displayed?" />
  <onentry id="20161121029040" type="displayed?" />
  <onentry id="20161121029041" type="contains" />
  <onentry id="20161121029042" type="displayed?" />
  <onentry id="20161121029043" type="enabled?" />
  <onentry id="20161121029047" type="not_displayed?" />
  <transition id="20161121029012" target="20161121029011" />
  <state id="save␣query" type="form">
    <send label="s20161121425003" type="required" element="checkbox" />
    <send label="s20161121425004" type="required" />
    <send label="s20161121425005" type="required" />
    <transition type="ajax" label="20161121425002">
      <submit target="20161121029010" />
      <error target="20161131117001" />
    </transition>
    <onexit id="20161121425006" type="displayed?" />
    <onexit id="2017031127005" type="not_displayed?" />
  </state>
</state>
<state id="20161121029011">
  <onentry id="20161121425013" type="contains" />
  <onentry id="20161121425014" type="enabled?" />
  <transition id="20161121029014" target="20161121029001" />
  <state id="alert␣delete␣query" type="form">
    <send label="s20161121425017" type="required" element="checkbox" />
    <transition type="alert" label="20161121425016">
      <submit target="20161121029011" />
    </transition>
    <onexit id="20161121425018" type="displayed?" />
  </state>
</state>
<state id="20161121029015">
  <onentry id="20161121425007" type="displayed?" />
  <onentry id="20161121425008" type="default" />
  <onentry id="20161121425009" type="enabled?" />
  <onentry id="20161121425010" type="is_selected" />
  <onentry id="20161121425011" type="enabled?" />
  <onentry id="20161121425028" type="is_not_selected" />
</state>
<state id="20161121029017">
```

```
    <onentry id="20161121425019" type="displayed?" />
    <onentry id="20161121425020" type="attribute" />
    <onentry id="20161121425021" type="enabled?" />
  </state>
  <state id="20161121029020">
    <onentry id="20161121425025" type="displayed?" />
    <onentry id="20161121425026" type="enabled?" />
  </state>
  <state id="20161121851003">
    <onentry id="20161121851005" type="displayed?" />
  </state>
  <state id="20161121851006">
    <onentry id="20161121851007" type="displayed?" />
    <onentry id="20161121851008" type="contains" />
  </state>
  <state id="20161131049001">
    <onentry id="20161131049002" type="displayed?" />
    <onentry id="20161131049003" type="contains" />
  </state>
  <state id="20161131049004">
    <onentry id="20161131049006" type="displayed?" />
  </state>
  <state id="20161131117001">
    <onentry id="20161131117002" type="displayed?" />
    <onentry id="20161131117003" type="contains" />
  </state>
</scxml>
```


A.1.2 *Mapeamento*


```
{
  "2017031127001": {
    "how_to_find": "className",
    "what_to_find": "header-full-title",
    "what_to_do": "getText"
  },
  "2017031127002": {
    "how_to_find": "linkText",
    "what_to_find": "Home",
    "what_to_do": "getCssValue",
    "type_of_action": "background-color"
  },
  "2017031127003": {
    "how_to_find": "className",
    "what_to_find": "dropdown-toggle",
    "what_to_do": "getText"
  },
  "20161141559012": {
    "how_to_find": "linkText",
```

```
    "what_to_find": "About␣us",
    "what_to_do": "click"
  },
  "20161141559017": {
    "how_to_find": "cssSelector",
    "what_to_find": "#header-full-top␣>␣DIV:nth-child(1)␣>␣NAV:nth-child(2)␣>␣DIV:nth-
        child(1)␣>␣A:nth-child(1)",
    "what_to_do": "click"
  },
  "2016114208001": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)",
    "what_to_do": "getText"
  },
  "s20161141559032": {
    "how_to_find": "className",
    "what_to_find": "dropdown-toggle",
    "what_to_do": "click"
  },
  "s20161141559033": {
    "how_to_find": "name",
    "what_to_find": "user[email]",
    "what_to_do": "sendKeys"
  },
  "s20161141559034": {
    "how_to_find": "name",
    "what_to_find": "user[password]",
    "what_to_do": "sendKeys"
  },
  "s20161141559035": {
    "how_to_find": "className",
    "what_to_find": "translation_missing",
    "what_to_do": "click"
  },
  "20161141559037": {
    "how_to_find": "cssSelector",
    "what_to_find": "#header-full-top␣>␣DIV:nth-child(1)␣>␣NAV:nth-child(2)␣>␣DIV:nth-
        child(2)␣>␣DIV:nth-child(2)␣>␣FORM:nth-child(1)␣>␣BUTTON:nth-child(6)",
    "what_to_do": "submit"
  },
  "20161141559013": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
        child(1)␣>␣H2:nth-child(1)",
    "what_to_do": "getText"
  },
  "20161141559014": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
        child(2)",
    "what_to_do": "getText"
```

```
},
"20161141559015": {
  "how_to_find": "id",
  "what_to_find": "default",
  "what_to_do": "getText"
},
"20161141559026": {
  "how_to_find": "className",
  "what_to_find": "section-title",
  "what_to_do": "getText"
},
"20161141559027": {
  "how_to_find": "linkText",
  "what_to_find": "Log␣in",
  "what_to_do": "getCssValue",
  "type_of_action": "color"
},
"20161141559028": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)",
  "what_to_do": "getText"
},
"s20161141559019": {
  "how_to_find": "name",
  "what_to_find": "user[name]",
  "what_to_do": "sendKeys"
},
"s20161141559020": {
  "how_to_find": "name",
  "what_to_find": "user[email]",
  "what_to_do": "sendKeys"
},
"s20161141559021": {
  "how_to_find": "name",
  "what_to_find": "user[password]",
  "what_to_do": "sendKeys"
},
"s20161141559022": {
  "how_to_find": "name",
  "what_to_find": "user[password_confirmation]",
  "what_to_do": "sendKeys"
},
"20161141559024": {
  "how_to_find": "cssSelector",
  "what_to_find": "#new_user␣>␣BUTTON:nth-child(7)",
  "what_to_do": "submit"
},
"2016114208002": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)",
  "what_to_do": "getText"
```

```
},
"2016114208003": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(2)␣>␣DIV:nth-
      child(1)␣>␣H1:nth-child(1)",
  "what_to_do": "getText"
},
"2016114208004": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣H1:nth-child(1)",
  "what_to_do": "getText"
},
"2016114208007": {
  "how_to_find": "linkText",
  "what_to_find": "Welcome␣admin",
  "what_to_do": "getText"
},
"2016114208009": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(2)␣>␣DIV:nth-
      child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child(1)␣>␣I:nth-child(1)",
  "what_to_do": "click"
},
"20161121029002": {
  "how_to_find": "linkText",
  "what_to_find": "Periodic␣Table",
  "what_to_do": "click"
},
"20161121029018": {
  "how_to_find": "cssSelector",
  "what_to_find": "#header-full-top␣>␣DIV:nth-child(1)␣>␣NAV:nth-child(2)␣>␣UL:nth-
      child(2)␣>␣LI:nth-child(1)␣>␣A:nth-child(1)␣>␣I:nth-child(1)",
  "what_to_do": "click"
},
"2016114208019": {
  "how_to_find": "xpath",
  "what_to_find": "//*[@id='bs-example-navbar-collapse-1']/ul/li[3]",
  "what_to_do": "getText",
  "type_of_action": "class"
},
"2016114208020": {
  "how_to_find": "name",
  "what_to_find": "ontology[file]",
  "what_to_do": "getText"
},
"2016114208021": {
  "how_to_find": "className",
  "what_to_find": "section-title",
  "what_to_do": "getText"
},
"s2016114208010": {
```

```
    "how_to_find": "name",
    "what_to_find": "ontology[name]",
    "what_to_do": "sendKeys"
  },
  "s2016114208011": {
    "how_to_find": "id",
    "what_to_find": "ontology_desc",
    "what_to_do": "sendKeys"
  },
  "s2016114208012": {
    "how_to_find": "cssSelector",
    "what_to_find": "#new_ontology␣>␣DIV:nth-child(5)␣>␣LABEL:nth-child(1)",
    "what_to_do": "click"
  },
  "s2016114208014": {
    "how_to_find": "name",
    "what_to_find": "ontology[file]",
    "what_to_do": "sendKeys"
  },
  "2016114208016": {
    "how_to_find": "cssSelector",
    "what_to_find": "#new_ontology␣>␣BUTTON:nth-child(7)",
    "what_to_do": "submit"
  },
  "2016114208018": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)",
    "what_to_do": "getCssValue",
    "type_of_action": "background-color"
  },
  "20161121029022": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(2)␣>␣DIV:nth-
        child(3)␣>␣TABLE:nth-child(2)␣>␣TBODY:nth-child(2)␣>␣TR:nth-child(1)␣>␣TD:nth-
        child(2)",
    "what_to_do": "getText"
  },
  "20161121029023": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(2)␣>␣DIV:nth-
        child(1)␣>␣DL:nth-child(2)␣>␣DD:nth-child(8)",
    "what_to_do": "getText"
  },
  "20161121029024": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(2)␣>␣DIV:nth-
        child(1)␣>␣DL:nth-child(4)␣>␣DD:nth-child(6)␣>␣A:nth-child(1)␣>␣BUTTON:nth-
        child(1)",
    "what_to_do": "getText"
  },
```

```
"20161121029025": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child(2)",
  "what_to_do": "getText"
},
"20161121029026": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(3)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child
      (1)",
  "what_to_do": "getText"
},
"20161121029027": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(2)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child
      (1)",
  "what_to_do": "getText"
},
"20161121029028": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣DL:nth-child(2)␣>␣DD:nth-child(2)",
  "what_to_do": "getText"
},
"20161121029029": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣DL:nth-child(2)␣>␣DD:nth-child(6)␣>␣FORM:nth-child(1)␣>␣BUTTON:nth-
      child(4)",
  "what_to_do": "getText"
},
"20161121029030": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣DL:nth-child(4)␣>␣DD:nth-child(2)",
  "what_to_do": "getText"
},
"20161121029031": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(2)␣>␣DIV:nth-child(2)␣>␣DIV:nth-child(1)",
  "what_to_do": "getText"
},
"20161121029016": {
  "how_to_find": "cssSelector",
```

```
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
        child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child(2)␣>␣I:nth-child(1)",
    "what_to_do": "click"
  },
  "20161121029019": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
        child(2)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child
        (1)",
    "what_to_do": "click"
  },
  "20161121029021": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
        child(3)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child
        (1)␣>␣I:nth-child(1)",
    "what_to_do": "click"
  },
  "2017031127004": {
    "how_to_find": "id",
    "what_to_find": "prefixes_add_all",
    "what_to_do": "getText"
  },
  "s20161121029003": {
    "how_to_find": "id",
    "what_to_find": "prefixes_select",
    "what_to_do": "click"
  },
  "s20161121029004": {
    "how_to_find": "id",
    "what_to_find": "prefixes_add",
    "what_to_do": "click"
  },
  "s20161121029005": {
    "how_to_find": "id",
    "what_to_find": "prefixes_add",
    "what_to_do": "click"
  },
  "s20161121029006": {
    "how_to_find": "id",
    "what_to_find": "prefixes_add",
    "what_to_do": "click"
  },
  "s20161121029007": {
    "how_to_find": "name",
    "what_to_find": "query[timeout]",
    "what_to_do": "sendKeys"
  },
  "s20161121029034": {
    "how_to_find": "id",
    "what_to_find": "prefixes_add",
    "what_to_do": "click"
```

```
},
"s20161131049005": {
  "how_to_find": "id",
  "what_to_find": "prefixes_add",
  "what_to_do": "click"
},
"20161121029009": {
  "how_to_find": "cssSelector",
  "what_to_find": "#submit_query␣>␣DIV:nth-child(5)␣>␣BUTTON:nth-child(2)",
  "what_to_do": "submit"
},
"20161121425027": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)",
  "what_to_do": "getText"
},
"s20161121425024": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child(3)␣>␣I:nth-child(1)",
  "what_to_do": "click"
},
"20161121425023": {
  "what_to_do": "accept"
},
"20161121029038": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child(1)␣>␣I:nth-child(1)",
  "what_to_do": "getText"
},
"20161121029039": {
  "how_to_find": "id",
  "what_to_find": "div_prefixes",
  "what_to_do": "getText"
},
"20161121029040": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(4)␣>␣DIV:nth-
      child(1)",
  "what_to_do": "getText"
},
"20161121029041": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(4)␣>␣DIV:nth-
      child(1)␣>␣H1:nth-child(1)␣>␣SMALL:nth-child(1)",
  "what_to_do": "getText"
},
"20161121029042": {
```

```
    "how_to_find": "className",
    "what_to_find": "col-md-3",
    "what_to_do": "getText"
  },
  "20161121029043": {
    "how_to_find": "id",
    "what_to_find": "query_start_save",
    "what_to_do": "getText"
  },
  "20161121029047": {
    "how_to_find": "id",
    "what_to_find": "query_saving",
    "what_to_do": "getText"
  },
  "20161121029012": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
        child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child(1)␣>␣I:nth-child(1)",
    "what_to_do": "click"
  },
  "20161121425006": {
    "how_to_find": "cssSelector",
    "what_to_find": "#query_save_area␣>␣DIV:nth-child(1)",
    "what_to_do": "getText"
  },
  "2017031127005": {
    "how_to_find": "id",
    "what_to_find": "query_stop_save",
    "what_to_do": "getText"
  },
  "s20161121425003": {
    "how_to_find": "id",
    "what_to_find": "query_start_save",
    "what_to_do": "click"
  },
  "s20161121425004": {
    "how_to_find": "id",
    "what_to_find": "name",
    "what_to_do": "sendKeys"
  },
  "s20161121425005": {
    "how_to_find": "id",
    "what_to_find": "desc",
    "what_to_do": "sendKeys"
  },
  "20161121425002": {
    "how_to_find": "id",
    "what_to_find": "query_save",
    "what_to_do": "click"
  },
  "20161121425013": {
    "how_to_find": "linkText",
    "what_to_find": "Periodic␣Table",
    "what_to_do": "getText"
```

```
},
"20161121425014": {
  "how_to_find": "cssSelector",
  "what_to_find": "TABLE:nth-child(1)␣>␣TBODY:nth-child(2)␣>␣TR:nth-child(1)␣>␣
      TD:nth-child(3)␣>␣A:nth-child(1)",
  "what_to_do": "getText"
},
"20161121029014": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child(1)",
  "what_to_do": "click"
},
"20161121425018": {
  "how_to_find": "cssSelector",
  "what_to_find": "SECTION:nth-child(3)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣
      DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)",
  "what_to_do": "getText"
},
"s20161121425017": {
  "how_to_find": "cssSelector",
  "what_to_find": "TABLE:nth-child(1)␣>␣TBODY:nth-child(2)␣>␣TR:nth-child(1)␣>␣
      TD:nth-child(3)␣>␣A:nth-child(1)",
  "what_to_do": "click"
},
"20161121425016": {
  "what_to_do": "accept"
},
"20161121425007": {
  "how_to_find": "className",
  "what_to_find": "section-title",
  "what_to_do": "getText"
},
"20161121425008": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣H3:nth-child(2)",
  "what_to_do": "getText"
},
"20161121425009": {
  "how_to_find": "id",
  "what_to_find": "ontology_desc",
  "what_to_do": "getText"
},
"20161121425010": {
  "how_to_find": "id",
  "what_to_find": "ontology_public",
  "what_to_do": "getText"
},
"20161121425011": {
  "how_to_find": "cssSelector",
  "what_to_find": "FORM␣>␣BUTTON:nth-child(7)",
  "what_to_do": "getText"
},
```

```
"20161121425028": {
  "how_to_find": "name",
  "what_to_find": "ontology[shared]",
  "what_to_do": "getText"
},
"20161121425019": {
  "how_to_find": "className",
  "what_to_find": "section-title",
  "what_to_do": "getText"
},
"20161121425020": {
  "how_to_find": "name",
  "what_to_find": "user[name]",
  "what_to_do": "getText",
  "type_of_action": "value"
},
"20161121425021": {
  "how_to_find": "cssSelector",
  "what_to_find": "#edit_user␣>␣BUTTON:nth-child(9)",
  "what_to_do": "getText"
},
"20161121425025": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)",
  "what_to_do": "getText"
},
"20161121425026": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣H1:nth-child(1)␣>␣A:nth-child(1)",
  "what_to_do": "getText"
},
"20161121851005": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)",
  "what_to_do": "getText"
},
"20161121851007": {
  "how_to_find": "cssSelector",
  "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
      child(1)␣>␣DIV:nth-child(1)",
  "what_to_do": "getText"
},
"20161121851008": {
  "how_to_find": "className",
  "what_to_find": "lead",
  "what_to_do": "getText"
},
"20161131049002": {
```

```
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)",
    "what_to_do": "getText"
  },
  "20161131049003": {
    "how_to_find": "className",
    "what_to_find": "lead",
    "what_to_do": "getText"
  },
  "20161131049006": {
    "how_to_find": "cssSelector",
    "what_to_find": "#sb-site␣>␣DIV:nth-child(1)␣>␣SECTION:nth-child(3)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-child(1)␣>␣DIV:nth-
        child(1)␣>␣DIV:nth-child(1)",
    "what_to_do": "getText"
  },
  "20161131117002": {
    "how_to_find": "cssSelector",
    "what_to_find": "#query_save_area␣>␣DIV:nth-child(1)",
    "what_to_do": "getText"
  },
  "20161131117003": {
    "how_to_find": "cssSelector",
    "what_to_find": "#query_save_area␣>␣DIV:nth-child(1)␣>␣UL:nth-child(2)␣>␣LI:nth-
        child(1)",
    "what_to_do": "getText"
  }
}
```

A.1.3 *Valores*

```
[
  { "2017031127002" : "rgba(0,␣153,␣218,␣1)" },
  { "2017031127003" : "Login" },
  { "2016114208001" : "successfully" },
  { "s20161141559032" : "" },
  { "s20161141559033" : "admin@admin" },
  { "s20161141559034" : "1234567890" },
  { "s20161141559035" : "" },
  { "20161141559013" : "About␣project" },
  { "20161141559015" : "http://localhost:3000/about" },
  { "20161141559027" : "rgba(0,␣153,␣218,␣1)" },
  { "20161141559028" : "A␣message␣with␣a␣confirmation␣link␣has␣been␣sent␣to␣your␣email
      ␣address.␣Please␣follow␣the␣link␣to␣activate␣your␣account." },
  { "s20161141559019" : "12345" },
  { "s20161141559020" : "teste@teste" },
  { "s20161141559021" : "1234567890" },
```

```
  { "s20161141559022" : "1234567890" },
  { "2016114208003" : "My␣ontologies" },
  { "2016114208004" : "Public␣ontologies" },
  { "2016114208007" : "Welcome␣admin" },
  { "2016114208019" : "active" },
  { "s2016114208010" : "Periodic␣Table" },
  { "s2016114208011" : "Periodic␣table␣ontology" },
  { "s2016114208012" : "" },
  { "s2016114208014" : "/Users/miguelpinto/Dropbox/4.Ano/Ontologias/PeriodicTable.owl"
      },
  { "2016114208018" : "rgba(221,␣255,␣239,␣1)" },
  { "20161121029022" : "was␣created" },
  { "20161121029023" : "admin" },
  { "20161121029028" : "Periodic␣Table" },
  { "20161121029030" : "Yes" },
  { "s20161121029003" : "PREFIX␣rdf:␣<http://www.w3.org/1999/02/22-rdf-syntax-ns#>" },
  { "s20161121029004" : "" },
  { "s20161121029005" : "" },
  { "s20161121029006" : "" },
  { "s20161121029007" : "60" },
  { "s20161121029034" : "" },
  { "s20161131049005" : "" },
  { "s20161121425024" : "" },
  { "20161121029041" : "11" },
  { "s20161121425003" : "" },
  { "s20161121425004" : "All␣classes" },
  { "s20161121425005" : "Get␣all␣the␣classes␣of␣this␣ontology" },
  { "20161121425013" : "Table" },
  { "s20161121425017" : "" },
  { "20161121425008" : "Periodic␣Table" },
  { "20161121425020" : "admin" },
  { "20161121851008" : "error" },
  { "20161131049003" : "error" },
  { "20161131117003" : "Name" }
]
```


A.1.4 *Mutações*


```
[
  {
    "type" : "lapse",
    "model_element": "s20161141559035",
    "fail" : "0"
  },
  {
    "type" : "slip",
    "model_element": "s20161141559019",
    "fail" : "0"
  },
```

```
  {
    "type" : "mistake",
    "model_element": "s20161141559022",
    "value" : "12345",
    "fail" : "1"
  },
  {
    "type" : "mistake",
    "model_element": "s2016114208011",
    "value" : "The␣periodic␣table␣ontology",
    "fail" : "0"
  },
  {
    "type" : "lapse",
    "model_element": "s2016114208012",
    "fail" : "0"
  },
  {
    "type" : "lapse",
    "model_element": "s2016114208014",
    "fail" : "1"
  },
  {
    "type" : "slip",
    "model_element": "s20161121029005",
    "fail" : "0"
  },
  {
    "type" : "slip",
    "model_element": "s20161131049005",
    "fail" : "0"
  },
  {
    "type" : "lapse",
    "model_element": "s20161121425004",
    "fail" : "1"
  },
  {
    "type" : "mistake",
    "model_element": "s20161121425005",
    "value" : "All␣the␣existent␣class␣for␣this␣ontology",
    "fail" : "0"
  }
]
```