4. Metodologia

Este capítulo apresenta a pesquisa e as decisões de projeto relacionadas ao assunto da dissertação.

4.1.Visão geral

A Figura 1 apresenta o processo construído, que envolve desde a descrição textual dos casos de uso à análise do atendimento do software sob teste (SST) a esta especificação.

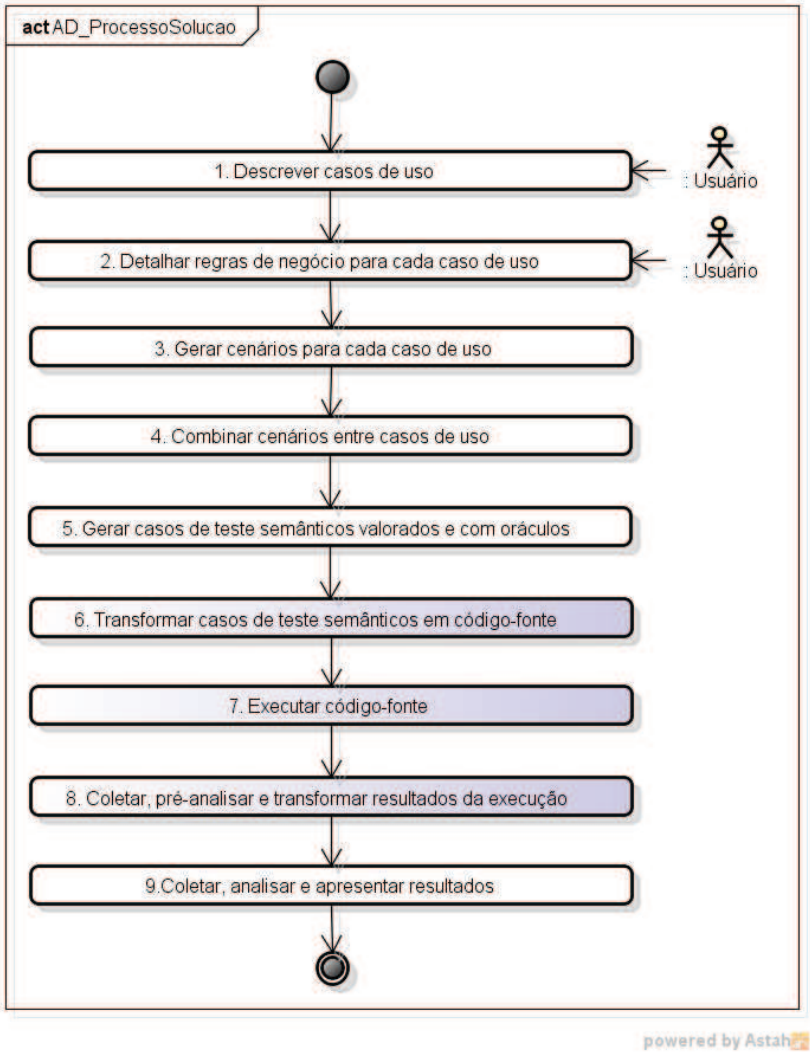

Figura 1 - Processo realizado pela ferramenta construída

Nesse processo, temos as seguintes etapas:

1. **Descrever casos de uso**: Preenchimento da descrição textual dos casos de uso, realizada pelo usuário;
2. **Detalhar regras de negócio para cada caso de uso**: Detalhamento realizado pelo usuário, onde são pormenorizadas as regras de negócio relacionadas aos passos realizados nos fluxos dos casos de uso;
3. **Gerar cenários para cada caso de uso**: Utiliza a informação dos fluxos e seus passos para gerar a combinação dos caminhos que podem ser percorridos na execução do caso de uso, de acordo com o documentado;
4. **Combinar cenários entre casos de uso**: Utiliza a informação das pré-condições para analisar dependências de estados entre casos de usos e também substitui a chamada a casos de uso, realizada por passos de fluxos, pela chamada a cenários desses casos de uso;
5. **Gerar casos de teste semânticos valorados e com oráculos**: Usa a informação dos cenários e do detalhamento das regras de negócio para gerar casos de teste semânticos valorados e seus oráculos;
6. **Transformar casos de teste semânticos em código-fonte**: Transforma para a linguagem de programação e *frameworks* de testes escolhidos;
7. **Executar código-fonte**: Realiza a execução do código-fonte gerado para testar a funcionalidade do SST, verificando sua adesão à especificação. Esta execução gera um arquivo de *log* de execução, no formato particular do *framework* de testes escolhido;
8. **Coletar, pré-analisar e transformar resultados da execução**: Coleta os resultados da execução dos testes, através do *log* de execução gerado pelo *framework* de testes, realiza uma pré-análise de falhas e erros encontrados, de acordo com a instrumentação do código realizada e transforma os resultados para um formato independente de *frameworks*, para que seja lido pela ferramenta;
9. **Coletar, analisar e apresentar resultados**: Analisa e apresenta os

resultados da execução dos testes, em relação às expectativas.

As etapas 1, 2, 3, 4, 5 e 9 são realizadas pela ferramenta em si, enquanto as etapas 6, 7 e 8 são realizadas por extensões da aplicação, para a linguagem e *framework* de testes alvo. Todas elas serão detalhadas a seguir.

4.2.Descrição textual de casos de uso (Etapa 1)

A Tabela 7 apresenta os campos utilizados pela ferramenta na descrição textual de casos de uso, com uma indicação se são utilizados na geração de testes.

Tabela 7 - Campos usados na descrição textual de casos de uso

Campo	Usado para gerar testes
Nome	Não
Objetivo	Não
Importância	Sim
Atores	Não
Pré-condições	Sim
Acessível Diretamente	Sim
Fluxos Disparadores	Sim
Fluxo Principal	Sim
Fluxos Alternativos	Sim
Regras de Negócio	Sim
Artefatos Correlatos	Não

A seção 6.7 fornece exemplos práticos da aplicação desta descrição textual. O significado de cada um desses campos, em complemento ao exposto na seção 2.1, é descrito a seguir.

- **Acessível diretamente**: Estabelece se o caso de uso é acessível diretamente, ou seja, se há alguma opção na tela principal do SST que permita acessá-lo. Este dado é importante para que não sejam gerados cenários para testar o caso de uso isoladamente, uma vez que não há meios de acessá-lo diretamente;
- **Pré-condições**: Serão utilizadas como meio de controlar os estados do sistema, estabelecendo as dependências entre casos de uso, verifi-

cando a presença dos estados no momento da execução e realizando a chamada dos casos de uso que geram os estados necessários. Logo, ao invés das pré-condições serem transformadas em assertivas sobre propriedades presentes na interface com o usuário ou em outros artefatos do sistema, podem ser transformadas em uma espécie de **máquina de estados**, verificando estados e ativando os casos de uso relacionados;

- **Pós-condições**: Como as pré-condições, serão utilizadas como meio de controlar os estados do sistema. Seguindo o exposto no trabalho de Kassel (2006), será declarada separadamente **para cada fluxo** do caso de uso, uma vez que há fluxos que não levam à conclusão correta do caso de uso e, consequentemente, não geram pós-condições e de que determinados fluxos geram pós-condições diferentes;
- **Importância**: Será calculada como o somatório da importância de seus fluxos (seção 4.2.2). Ela servirá como viés para selecionar um conjunto de casos de uso para compor a execução de testes rápidos.

As subseções a seguir descrevem em detalhes a estrutura dos fluxos de casos de uso. A seção 4.3 (etapa 2 do processo) descreve, em detalhes, a estrutura das regras de negócio.

4.2.1.Fluxos Disparadores (FD)

São fluxos em que ocorre a interação do ator com o sistema para que o caso de uso inicie. Ocorrem antes do Fluxo Principal (descrito a seguir) e são usados apenas quando o caso de uso é chamado diretamente (através da tela inicial do sistema, por exemplo), e não através de outro caso de uso.

Usualmente um caso de uso possui apenas um FD, que representa, por exemplo, a interação do ator com o sistema através de sua tela principal, acessando a opção que dispara o caso de uso através de *menus*. Porém, a ferramenta permite a definição de mais de um FD, para possibilitar o início do caso de uso por outros meios, como ícones ou teclas de atalho, por exemplo.

Cada fluxo possui uma sequência de passos, detalhada na Seção 4.2.4.

4.2.2.Fluxo Principal (FP)

No Fluxo Principal serão informados os seguintes campos, a serem usados no processo de geração de testes:

- **Prioridade de construção**: Prioridade de construção do fluxo, em relação aos outros fluxos do sistema. Varia de 1 a 4, representando a análise MoSCoW (*Must*, *Should*, *Could*, *Won't*) proposta pelo

(INTERNATIONAL INSTITUTE OF BUSINESS ANALYSIS, 2009) e sendo adaptado como 1="Não agora, mas gostaria no futuro", 2="Poderia, se não afetasse o resto", 3="Necessário", 4="Fundamental". Este campo pode ser utilizado pela equipe de desenvolvimento para se orientar quanto à ordem de construção do fluxo em relação ao sistema, facilitando a criação de um *backlog* no estilo usado no Scrum (SUTHERLAND & SCHWABER, 1995). Para a geração dos testes, comporá o cálculo de **Importância** do fluxo (abaixo);

- **Complexidade**: Complexidade do fluxo, estimada pela equipe de desenvolvimento, variando de 1 a 5, sendo 1="Muito Pequena", 2="Pequena", 3="Média", 4="Grande" e 5="Muito Grande". Serve como meio de estimar o esforço necessário para construir o fluxo. Para a geração dos testes, comporá o cálculo de **Importância** (abaixo);
- **Frequência de Uso**: Frequência de uso do fluxo. Varia de 1 a 3, sendo 1="Quase nunca", 2="Normal" e 3="Muito usado". Para a geração dos testes, comporá o cálculo de **Importância** do fluxo (abaixo);
- **Importância** (I): Será calculada com base na Prioridade de Construção (P), na Complexidade (C) e na Frequência de Uso (F), como $I = P * C * F$. Ela poderá ser usada, futuramente, como viés para seleção do conjunto de fluxos que comporá a execução de testes rápidos;
- **Sequência de Passos:** Declarações dos passos a serem executados pelo ator ou pelo sistema para realizar o feito esperado pelo fluxo. Sua sintaxe é detalhada adiante.

4.2.3.Fluxos Alternativos (FA)

Conjunto de fluxos preenchidos, cada um, de forma semelhante ao Fluxo Principal. Além dos mesmos campos possuídos pelo FP, possui também os campos descritos a seguir.

- **Descrição**: Texto que identifica a função do Fluxo Alternativo (não usado para testes, obviamente);

- **Fluxos Alternativos Não Influenciadores**: Permite indicar quais outros Fluxos Alternativos não influenciam sua execução, ou seja, quais fluxos que ao serem executados não alteram (mesmo que levemente) algum estado do fluxo atual. Este campo influencia no processo de **geração de cenários**, podendo diminuir o número de cenários gerados (ver seção 4.4);

- **Tipo**: O presente trabalho classifica os Fluxos Alternativos quanto ao seu *tipo*, da seguinte forma:

    1. **Fluxo Terminador (FT)**: É um fluxo alternativo que leva à conclusão do caso de uso com sucesso (alcançando o benefício gerado por ele). Possui os seguintes campos:

        1.1. **Fluxo Iniciador**: É o fluxo (FP ou FA) em que ele é iniciado.

        1.2. **Passo Iniciador**: É o passo do Fluxo Iniciador em que ele é disparado.

    2. **Fluxo Retornável (FR)**: É um fluxo alternativo que, após executar seus passos, retorna para um determinado fluxo, geralmente o que o iniciou. Possui os mesmos campos de um Fluxo Terminador, bem como:

        2.1. **Fluxo de Retorno**: Fluxo para onde irá retornar;

        2.2. **Passo de Retorno**: Passo do Fluxo de Retorno para o qual irá retornar.

    3. **Fluxo Cancelador (FC)**: É um fluxo alternativo de exceção, disparado pelo usuário a qualquer momento durante a execução do caso de uso, que possui o objetivo de interromper e terminar sua execução, sem que o benefício gerado pelo caso de uso possa ser alcançado. Não possui campos adicionais.

    4. **Fluxo de Exceção Recuperável (FER)**: É um fluxo alternativo de exceção, disparado pelo sistema, onde há a possibilidade de se recuperar (da exceção) e voltar para o fluxo onde ela ocorreu. Possui os mesmos campos de um Fluxo Retornável, porém os campos que são adicionais (Fluxo Iniciador, Passo Iniciador e Passo de Retorno)

são de preenchimento *opcional*, uma vez que o momento de ocorrência de uma determinada exceção pode não ser previsível. O FER **não será usado para a geração de testes**, uma vez que simular as condições (internas ou externas) necessárias para a ocorrência da exceção pode não ser possível através do uso de testes funcionais. É importante observar, entretanto, que, no presente trabalho, o não atendimento às regras de negócio dos campos da interface com o usuário *não* são vistas como uma exceção (e, portanto não geram FERs).

5. **Fluxo de Exceção Irrecuperável (FEI)**: É um fluxo alternativo de exceção, disparado pelo sistema, onde não há a possibilidade de se recuperar (da exceção), levando ao término do caso de uso, sem sucesso. Possui os mesmos campos de um Fluxo Terminador, porém os campos que são adicionais (Fluxo Iniciador e Passo Iniciador) são de preenchimento *opcional*, uma vez que o momento da ocorrência da exceção pode não ser previsível. O FEI **não será usado para a geração de testes**, uma vez que simular as condições (internas ou externas) necessárias para a ocorrência da exceção pode não ser possível através do uso de testes funcionais.

- **Sequência de Passos**: Idem Fluxo Principal.

4.2.4.Detalhes sobre os passos dos fluxos

O presente trabalho classifica as ações executadas num passo, tanto pelo ator quanto pelo sistema, da seguinte forma:

1. **Interação com Elemento**: Quando o sistema ou ator realiza alguma operação sobre um elemento de interface gráfica (*widget*). Por exemplo, quando o sistema exibe uma janela, ou quando o usuário clica num botão;

2. **Verificação de Elemento**: Quando o sistema realiza uma verificação sobre as regras de negócio de um elemento editável (que pode receber valores do usuário, através de entrada de dados);

3. **Chamada a Caso de Uso**: Quando um caso de uso necessita ser chamado pelo sistema;
4. **Documentação**: Quando a ação realizada pelo sistema não é um evento relacionado à interface com o usuário (apesar de seu processamento poder alterar o status da mesma), servindo apenas como um meio de descrever (documentar) uma ou mais operações que devem ser realizadas por ele, internamente. Por exemplo, a geração de um relatório, ou a realização de um *download* de um arquivo.

4.2.4.1.Sintaxe de um passo

A Listagem 1, a seguir, apresenta a sintaxe (BNF) de um passo de um fluxo de um caso de uso.

```
<passo>        ::=    <disparador> <ação> <alvo>+
                |     <disparador> <documentação>
<disparador>   ::=    "ator" | "sistema"
<alvo>         ::=    <elemento> | <caso-de-uso>
<elemento>     ::=    <widget> | <URL> | <comando> | <tecla> | <tempo>

<ação>         ::=    string
<documentação> ::=    string
<caso-de-uso>  ::=    string
<widget>       ::=    string
<URL>          ::=    string
<comando>      ::=    string
<tecla>        ::=    string
<tempo>        ::=    integer
```

Listagem 1 - Sintaxe de um passo de um fluxo

Portanto o `ator` ou o `sistema` disparam uma `ação` sobre um ou mais `alvo`s, ou sobre uma `documentação`. Cada `alvo` pode ser um `elemento` ou um `caso de uso`. E o

`elemento` pode ser um `widget`, uma `URL`, um `comando`, uma `tecla` ou um `tempo` (em milissegundos, que é geralmente usado para aguardar um processamento).

O tipo de `alvo` e o número de `alvos` possíveis para uma `ação` podem variar conforme a configuração do vocabulário do perfil usado (a definição de um perfil é descrita na seção a seguir). O tipo de `elemento` a ser usado também pode variar conforme a `ação` escolhida.

4.2.4.2.Uso de perfis

A sintaxe mostrada anteriormente permite descrever boa parte das ações que um ator ou o sistema podem executar e concede certa flexibilidade para adaptação a diferentes *frameworks* de teste.

A ferramenta construída neste trabalho funciona com uso de **extensões**, que realizam a conversão de testes semânticos para a linguagem e *framework* de testes alvo. Cada extensão define um **perfil**, que corresponde às ações e tipos de elementos (*widgets*) com o qual o *framework* trabalha.

Apesar do perfil esperado pela extensão ser fixo e descrito num único idioma, é possível definir um **vocabulário**, com palavras correspondentes a cada ação do perfil, no idioma desejado. Estas palavras são chamadas de **apelidos**. Por exemplo, a ação "*click*" (que representa a ação de um clique) pode ter um apelido "clica em" (em Português do Brasil).

4.3.Detalhamento das regras de negócio (Etapa 2)

Este trabalho introduziu a possibilidade de descrever as regras de negócio (RN) para elementos usados nos passos de um fluxo (a seção 4.2.4.1 define um elemento). Com base nessas definições, é possível:

a) Conhecer o que representa um determinado elemento (seu significado), para saber o que fazer com ele. Se o elemento for editável, por exemplo, podendo receber entrada de dados, será possível descrevê-lo em mais detalhes;

b) Inferir os possíveis valores que podem ser recebidos por elementos editáveis e gerá-los automaticamente;

c) Conhecer a expectativa de comportamento do sistema quando uma regra é infringida pelo usuário e gerar oráculos automaticamente;

d) Realizar a verificação automática destas regras, através de testes funcionais;

e) Interromper a necessidade do usuário de descrever fluxos alternativos para verificar estas regras de negócio, uma vez que a ferramenta gerará automaticamente casos de teste para verificá-las.

Desta forma, a definição das regras de negócio ao mesmo tempo habilita a geração automática de testes e cria um benefício extra para a equipe de desenvolvimento, de não precisar descrever os vários fluxos alternativos que seriam necessários para tratar a verificação das regras de negócio.

4.3.1.Classificação de um elemento

A primeira forma de definir uma regra de negócio é classificando o elemento (seção 4.2.4.1):

a) **Tipo de elemento**: Identifica o tipo do elemento para fins de teste. Os tipos atualmente disponíveis são:

   i. **Widget**: Elemento de interface gráfica;

   ii. **URL**: Caminho de um recurso (ex.: de uma página *web*). Geralmente será usado para abrir um arquivo ou página *web*.

   iii. **Comando**: Comando a ser executado pelo sistema (ex.: `"C:\caminho\para\arquivo.exe"`). Difere semanticamente da URL para a ferramenta de geração de testes poder discernir o que deve ser feito com o arquivo passado;

   iv. **Combinação de teclas**: Permite processar uma tecla ou uma combinação delas (ex.: `CTRL + ALT + X`);

   v. **Tempo**: Permite definir um tempo de espera, em milissegundos, geralmente usado para aguardar a ocorrência de um evento.

   É possível ainda detalhar o tipo de *widget*, que podem variar de acordo com o perfil usado (seção 4.2.4.2). Por exemplo, pode haver um *widget*

do tipo *button*, outro *textbox*, outro *window*, e assim por diante.

b) **Editável**: Informação relacionada ao Tipo de Elemento (item a) e informa se um ator pode introduzir dados no elemento.

c) **Nome interno**: Nome do elemento para fins de teste. Enquanto o elemento receberá um nome legível, para ser apresentado na descrição do caso de uso, ele usará este nome interno para ser usado para identificá-lo durante o processo de testes. Por exemplo, um passo descrito como "Ator clica em OK", o OK pode ser um *widget* (botão) com nome interno "okButton".

Então, quando um elemento é classificado como **editável**, é possível definir:

d) **Tipo de valor**: Tipo de dado do valor aceito pelo elemento, podendo ser `String`, `Integer`, `Double`, `Boolean`, `Date`, `Time` e `DateTime`;

e) **Regra:** Possibilita informar uma ou mais faixas de valores, comprimentos, formatos, fonte de valores, etc., esperados. Atualmente as regras disponíveis são:

   i. **Obrigatório**: Permite estabelecer se o elemento tem seu preenchimento obrigatório;

   ii. **Valor Mínimo**: Permite estabelecer um valor mínimo para o elemento;

   iii. **Valor Máximo**: Permite estabelecer um valor mínimo para o elemento;

   iv. **Comprimento Mínimo**: Permite estabelecer um comprimento mínimo para o elemento (como `String`);

   v. **Comprimento Máximo**: Permite estabelecer um comprimento máximo para o elemento (como `String`);

   vi. **Expressão Regular**: Permite definir o formato esperado para o elemento, como uma expressão regular;

   vii. **Igual A**: Permite definir os possíveis valores aceitos para o elemento;

   viii. **Diferente De**: Permite definir os possíveis valores não aceitos

para o elemento.

f) **Mensagem:** É definida para cada Regra definida, sendo a mensagem esperada para quando o valor informando pelo usuário não atender a essa Regra.

Para cada **Regra**, com exceção da regra **Obrigatório**, é possível ainda definir se o valor será definido manualmente, obtido de outro elemento do mesmo caso de uso (existente antes do passo em que o elemento corrente se encontra), ou obtido através de uma consulta a um banco de dados (BD), conforme pode ser visto na Tabela 8, a seguir.

Tabela 8 - Fontes de valores para Regras de Negócio

Regra	Permite valor manual	Permite valor consultado (BD)	Permite valor de outro elemento
Valor Mínimo	Sim	Sim	Sim
Valor Máximo	Sim	Sim	Sim
Comprimento Mínimo	Sim	Sim	Sim
Comprimento Máximo	Sim	Sim	Sim
Expressão Regular	Sim	Não	Não
Igual A	Sim, mais de um	Sim	Sim
Diferente De	Sim, mais de um	Sim	Sim

Isto dá certa flexibilidade para a definição das regras e possibilita o uso de valores obtidos de bancos de dados confeccionados para teste usando dados com a mesma estrutura dos utilizados em produção.

Adicionalmente, os parâmetros das consultas a banco de dados podem receber valores advindos da mesma estrutura. Ou seja, uma consulta a banco de dados que possui uma restrição (*where*) dependente do valor de outro elemento, poderá recebê-lo no momento da geração dos testes.

A ferramenta mantém ainda cada resultado de consulta realizada em *cache*, permitindo que elementos que tenham a mesma consulta obtenham os mesmos

valores, além de acelerar a geração para os próximos elementos com a mesma regra.

4.3.2.Detalhamento da chamada a um caso de uso

A chamada a um caso de uso, realizada por um passo de um fluxo, deve ser detalhada para definir quais as *pós-condições* esperadas na execução deste caso de uso. Isto permite que apenas os cenários (do caso de uso chamado) que geram as pós-condições esperadas sejam selecionados para combinação. Do contrário, serão escolhidos os cenários em que o último fluxo é um Fluxo Principal ou um Fluxo Terminador (Seção 4.2.3). Este assunto é discutido em mais detalhes na Seção 4.5 (que trata da combinação de cenários).

4.3.3.Detalhamento de um oráculo

O detalhamento de um oráculo visa definir:

1. O **tipo do elemento gráfico** (*widget*) esperado para exibir a mensagem definida para quando um elemento receber um valor inválido (ex.: uma caixa de diálogo, um rótulo, etc.).
2. A **ocorrência esperada para exibição da mensagem**, que pode ser:
    a. Apenas uma mensagem, para o primeiro elemento verificado: visa tratar o caso de softwares que verificam um elemento de cada vez e exibem apenas uma mensagem, referente ao primeiro elemento com valor inválido;
    b. Apenas uma mensagem, para todos os elementos: visa tratar o caso de softwares que verificam todos os elementos e exibem uma única mensagem contendo todos os erros encontrados;
    c. Uma mensagem para cada elemento: visa tratar o caso de softwares que verificam um elemento de cada vez e exibem uma mensagem para cada elemento.[23]

[23] O autor deste trabalho considera esta uma má prática em relação à usabilidade da interface com o usuário, porém, não deixa de tratar este caso na ferramenta criada.

Por padrão (*default*), o *tipo de elemento gráfico* é uma caixa de diálogo e a *ocorrência da mensagem* é a definida na opção 2b (uma mensagem para todos os elementos).

4.4.Geração de cenários para cada caso de uso (Etapa 3)

Para cada caso de uso do sistema, será preciso realizar a combinação de seus fluxos (principal e alternativos), gerando os possíveis cenários de execução. Cada cenário partirá do fluxo principal e, possivelmente:

- Passará por um ou mais fluxos alternativos;
- Retornará ao fluxo principal; ou
- Cairá em recursão, repetindo passos anteriores.

Para ilustrar a formação dos cenários, consideremos o caso de uso exibido na Figura 2:

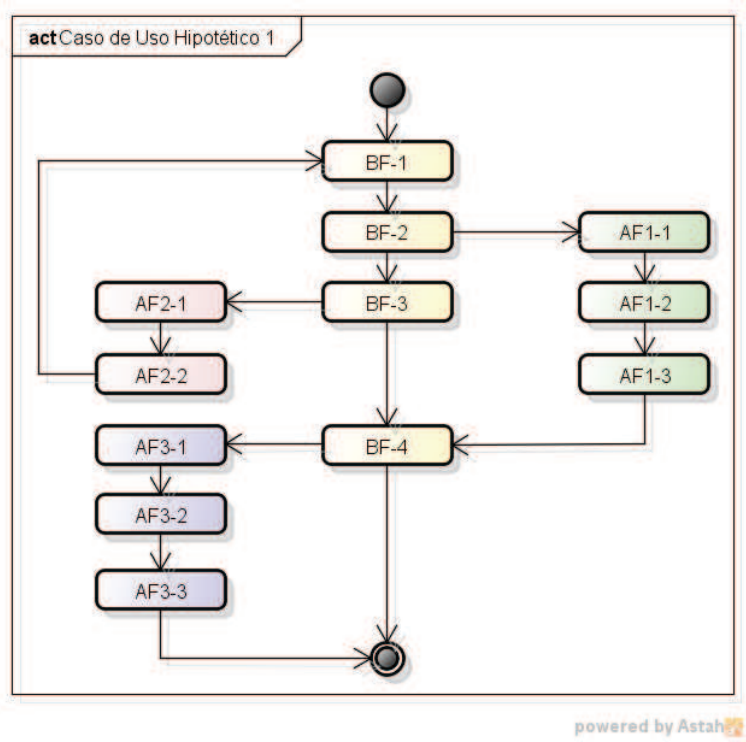

Figura 2 - Caso de Uso Hipotético 1

Neste caso de uso temos 4 fluxos:

- Um fluxo principal, BF;

- Um fluxo alternativo retornável, AF1;
- Um fluxo alternativo recursivo, AF2;
- Um fluxo alternativo terminável, AF3.

A cobertura de fluxos poderia ter sua combinação formada pela seguinte expressão.[24]

$$BF\big(* \,|AF1(BF)|AF2(BF)\big|AF3(*)\big)$$

A expressão parte do fluxo principal (BF) e segue para os demais fluxos na ordem dos passos. A combinação com asterisco indica que o fluxo pode ser percorrido sozinho. Assim, partindo de BF, teríamos as combinações: {BF}, {BF, AF1}, {BF, AF2} e {BF, AF3}. Porém, se analisarmos AF1 e AF2, veremos que eles retornam para BF, o que abre caminho para novas combinações. Estas combinações, entretanto, podem não ser possíveis na prática, pois é a ordem e o sentido dos passos que determinam os possíveis caminhos. Então, o algoritmo deve se basear no caminhamento dos **passos** e não na combinação dos fluxos.

Portanto, os possíveis cenários para o caso de uso hipotético exposto na Figura 2 seriam:

1. **{BF}**:

   BF-1, BF-2, BF-3, BF-4;

2. **{BF, AF1, BF}**:

   BF-1, AF1-1, AF1-2, AF1-3, BF-4;

3. **{BF, AF2, BF}**:

   BF-1, BF-2, AF2-1, AF2-2, BF-1, BF-2, BF-3, BF-4;

4. **{BF, AF3}**:

   BF-1, BF-2, BF-3, AF3-1, AF3-2, AF3-3;

5. **{BF, AF1, BF, AF3}**:

   BF-1, AF1-1, AF1-2, AF1-3, AF3-1, AF3-2, AF3-3;

[24] Os parênteses à direita do nome de cada fluxo determinam quais são as combinações possíveis deste fluxo. Cada fluxo combinável é separado por uma barra vertical (*pipe*). O asterisco representa que o fluxo é terminável, ou seja, encontra um fim.

6. **{BF, AF2, BF, AF1, BF}**:

   BF-1, BF-2, AF2-1, AF2-2, BF-1, AF1-1, AF1-2, AF1-3, BF-4;

7. **{BF, AF2, BF, AF1, BF, AF3}**:

   BF-1, BF-2, AF2-1, AF2-2, BF-1, AF1-1, AF1-2, AF1-3, AF3-1, AF3-2, AF3-3;

8. **{BF, AF2, BF, AF3}**:

   BF-1, BF-2, AF2-1, AF2-2, BF-1, BF-2, BF-3, AF3-1, AF3-2, AF3-3;

É interessante notar que ao existir um **passo** com um fluxo alternativo, se o fluxo alternativo for usado no cenário, este passo será substituído por todos os passos deste fluxo.

4.4.1.Casos de recursão

Diferente de algoritmos que levam em consideração a passagem por caminhos sem considerar a repetição de seus pontos, como o algoritmo de Complexidade Ciclomática (MCCABE, 1976) ou de Caminhamento em Profundidade (TARJAN, 1976), o algoritmo para geração de cenários deve considerá-la, pois a passagem por um fluxo pode influenciar o estado de outro, alterando seu comportamento e levando-o a trilhar um caminho diferente.

Nos casos de recursão, um fluxo, ao terminar, retorna para um passo anterior ao seu passo de início, fazendo com que seja possível o caminho ser repetido infinitas vezes. Se analisarmos a Figura 2, ou a expressão criada para ela, veremos que o fluxo alternativo AF2 retorna para o fluxo principal BF num caso de recursão. Neste exemplo simples, a recursão só permite variar a quantidade de passagens por AF2, mas não variar a passagem por outros fluxos. Num exemplo mais complexo, como o mostrado pela Figura 3 (onde os fluxos AF4 e AF5 são recursivos), a recursão possibilita uma grande variedade de novos caminhos.

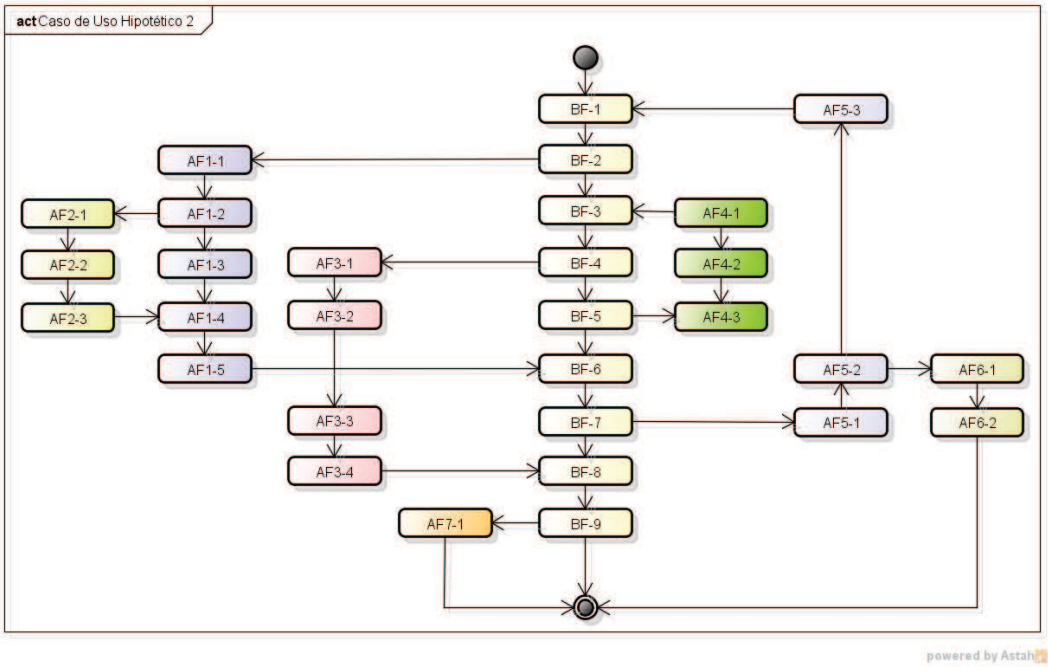

Figura 3 - Caso de Uso Hipotético 2

Não só a recursão potencializa o número de combinações, mas principalmente o número de vezes em que o fluxo recursivo é usado. Este número deve ser mantido baixo, para que não inviabilize a aplicação prática da geração de cenários.

4.4.1.1.Tratamento de casos de recursão

Na ferramenta construída por este trabalho, o número de passagens por um fluxo recursivo é parametrizável, possibilitando o impedimento de uma explosão combinatória. Para exemplificar, ao submeter a ela o caso de uso exemplificado na Figura 2, foram obtidos 12 e 16 cenários para, respectivamente, duas e três passagens. Já para o caso de uso da Figura 3, foram obtidos 309 e 1671 cenários para, respectivamente, duas e três passagens.

Desta forma, recomenda-se o uso padrão de duas passagens por fluxos recursivos (caso se queira verificar a passagens por eles mais de uma vez), a fim de viabilizar a geração de cenários.

4.4.2.Cobertura

A geração de cenários realizada cobre a passagem por todos os fluxos do caso de uso, bem como a combinação entre todos eles, pelo menos uma vez. Em cada fluxo, todos os passos são cobertos. Isto garante a observância de defeitos relacionados à passagem por determinados fluxos ou ocasionados por sua combinação.

Hassan & Yousif (2010) calculam a Cobertura de um Caso de Uso dividindo o *total de passos cobertos pelo teste* pelo *total de passos existentes no caso de uso*. Assim, se todos os passos são cobertos pelos testes, ter-se-á uma cobertura de 100% para o caso de uso, o que é feito pela presente ferramenta.

4.4.3.Algoritmo

A Listagem 2, a seguir, apresenta o algoritmo em pseudocódigo para a geração de cenários de um único caso de uso.

```
ALGORITHM GenerateScenarios( IN currentFlow, IN maxRepetitions,
  IN currentStep, IN currentScenario, OUT scenariosList )
BEGIN
  IF currentStep is NULL THEN
    currentStep <- first step IN currentFlow
  ENDIF
  FOR EACH step IN currentFlow STARTING AT currentStep DO
    FOR EACH flow IN leaving flows FROM step DO
      IF flow is returnable AND flow is recursive THEN
        returningStep <- returning step FROM flow
        count <- how many steps IN currentScenario are EQUAL TO returningStep
        IF count > maxRepetitions THEN
          CONTINUE
        ENDIF
      ENDIF
      newScenario <- COPY OF currentScenario
      CALL GenerateScenarios( flow, maxRepetitions, NULL, newScenario,
          scenarioList )
    ENDFOR
    ADD step TO currentScenario
  ENDFOR
  IF currentFlow is terminable THEN
    ADD currentScenario TO scenariosList
  ELSE IF currentFlow is returnable THEN
    returningFlow <- returning flow FROM currentFlow
    returningStep <- returning step FROM currentFlow
    CALL GenerateScenarios( returningFlow, maxRepetitions, returningStep,
        currentScenario, scenarioList )
  ENDIF
END
```

Listagem 2 - Algoritmo de geração de cenários de um caso de uso

Sua ideia geral é a de ir percorrendo cada passo de um fluxo e, quando for encontrado um fluxo saindo do passo, um novo cenário é criado para este caminho, recebendo os passos anteriores. Esse novo cenário é percorrido da mesma forma, procurando fluxos em cada passo. Quando um dos fluxos que saem de um passo for recursivo, é contado quantas vezes o passo para o qual o fluxo retorna já entrou no cenário. Se esta contagem já alcançou o limite definido (o número máximo de passagens por um fluxo recursivo), ele é simplesmente ignorado (não é criado um novo cenário para ele).

É importante observar que este algoritmo não realiza a combinação entre cenários, apenas gera os cenários para um caso de uso, sem considerar suas dependências (pré-condições) e eventuais chamadas para outros casos de uso. Ambos os casos serão tratados pelo algoritmo de combinação de cenários, descrito na próxima seção.

4.5.Combinação de cenários entre casos de uso (Etapa 4)

A combinação entre cenários de diferentes casos de uso, *A* e *B*, ocorrerá quando:

i. Algum passo de *A* **chamar** o caso de uso *B*;

ii. O caso de uso *A* possuir como **pré-condição** uma pós-condição gerada por *B*.

Em ambos os casos é estabelecida uma dependência de *A* para *B*. Porém, quando existe uma **dependência de estados** (pré-condição depender de pós-condição), os cenários de *B* devem ser gerados antes dos de *A*. Dessa forma, uma pós-condição de *B* servirá de entrada para *A*, num mecanismo semelhante a uma **máquina de estados**.

Considerando o SST, pode-se representar a rede de dependências entre seus vários casos de uso como um **grafo acíclico dirigido**. Aplicando uma ordenação topológica[25] (KAHN, 1962) desse grafo, é obtida a ordem correta para geração dos cenários.

Antes de combinar os cenários, entretanto, é preciso analisá-los, verificando se cenários predecessores geram os estados esperados pelos cenários sucessores. Usando a classificação dos fluxos realizada nas seções 4.2.1 e 4.2.3, é possível selecionar os cenários cujo *fluxo terminador* é classificado com um Fluxo Principal ou um Fluxo Terminador. Desses cenários, é possível selecionar apenas aqueles que geram as pós-condições desejadas, necessárias aos cenários sucessores. É para possibilitar essa seleção que o presente trabalho requer a definição de pós-condições por fluxo, e não por caso de uso, como comumente

[25] A ordenação topológica consiste em realizar uma ordenação linear num grafo acíclico dirigido de forma que para cada arco AB que vai do vértice A para o vértice B, A se torna *anterior* a B nesta ordenação.

encontrado na literatura.

Assim, tanto quando a combinação tiver de ser realizada porque *A chama B* como quando *A depender de B*, serão selecionados de *B* apenas os cenários que geram as pós-condições esperadas por *A* (chamada ou pré-condição).

Os cenários selecionados para cada caso são, porém, diferentes. No caso em que o caso de uso *A chama* o caso de uso *B*, deve ser escolhido de *B* os cenários ainda **sem a combinação com seus Fluxos Disparadores**. Isto ocorre porque *B* será chamado a partir de outro caso de uso, e não por sua execução "normal" ocorrida através da tela inicial do software.

4.5.1.Ordem da combinação de cenários

A combinação dos cenários é, então, realizada na seguinte ordem:

1. Cenários de casos de uso chamados por passos;
2. Cenários de fluxos disparadores;
3. Cenários de casos de uso de pré-condições.

Esta ordem garante que os cenários contenham o conteúdo correto ao longo do tempo. É importante lembrar que a geração respeita a ordem topológica dos casos de uso, evitando que sejam usados cenários incompletos (sem outras combinações necessárias).

4.5.2.Cobertura

Todos os cenários de cada caso e uso são verificados pela ferramenta. Em sua combinação com cenários de pré-condições ou de chamadas realizadas em passos, apenas os cenários que levam ao término com *sucesso* do caso de uso são selecionados. Isto porque se um cenário tiver de ser executado como pré-condição para certo caso de uso e este não obtiver sucesso, o sistema não poderá executar o caso de uso (ex.: se um caso de uso "Efetuar Login" for uma pré-condição e for selecionado um cenário que cancela sua execução, o caso de uso alvo não chegará a ser executado). Da mesma forma, quando um caso de uso realiza uma chamada a outro, em um de seus passos, caso o cenário deste outro não termine com

sucesso, é possível que o caso de uso atual também não o faça.

Por este motivo, dados todos os cenários de um caso de uso *A* (independente se terminam com sucesso ou não), cada um deles será combinado apenas com cenários de *B* que terminem com sucesso. Isto permite que a execução de todos os cenários de *A* chegue a um **estado terminal**.

4.5.2.1.Cálculo

Dados dois casos de uso quaisquer, A e B, do conjunto de casos de uso do software CDU, seja C_a o número de cenários do caso de uso A, C_b o número de cenários do caso de uso B, CS_b o número de cenários de sucesso de B, se A depende de B, então a cobertura dos cenários de A, CO_a, pode ser calculada como:

$$CO_a = \frac{CS_b \times C_a}{C_b \times C_a} = \frac{CS_b}{C_b}$$

Se por exemplo, um caso de uso B tiver 5 cenários, sendo 2 de sucesso, e outro caso de uso A, que depende de B, tiver 8 cenários, a cobertura total seria de 40 combinações, enquanto a cobertura alcançada seria de 16 combinações, ou 40% do total. Apesar desta cobertura não ser total (100%), acredita-se que ela seja eficaz para testes de casos de uso, pelos motivos expostos anteriormente.

4.5.2.2.Observação

É importante notar que a combinação entre cenários é multiplicativa, ou seja, a cada caso de uso adicionado, seu conjunto de cenários é multiplicado pelos atuais. Este trabalho considerou algumas soluções para este problema, detalhadas na seção 8.1. Estas soluções não puderam ser implementadas, por falta de tempo hábil.

4.5.3.Algoritmo

O algoritmo em pseudocódigo para a combinação de cenários é descrito na Listagem 3:

```
ALGORITHM CombineScenarios (IN useCaseList, IN OUT scenariosMap)
BEGIN
SORT useCaseList BY topological order
FOR EACH useCase IN useCaseList DO
  scenarios <- scenarios FROM useCase IN scenariosMap
  // 1. Combine use case calls
  scenariosThatHaveUseCaseCall <- scenarios that has use case calls FROM useCase
  combinedScenarioList <- CREATE scenario list
  FOR EACH scenario IN scenariosThatHaveUseCaseCall DO
    FOR EACH called use case FROM scenario DO
      scenariosWithSuccessfulEnd <- scenarios that have successful end
                                    IN called use case
      FOR EACH sucessfulScenario IN scenariosWithSuccessfulEnd DO
        newScenario <- COPY OF scenario that has use case calls
        REPLACE step that has use case call IN newScenario
                WITH steps FROM successfulScenario
        ADD newScenario TO combinedScenarioList
      ENDFOR
    ENDFOR
  ENDFOR
  REPLACE scenariosThatHaveUseCaseCall WITH combinedScenarioList IN useCase
  // 2. Combine activation flows with current scenarios
  combinedScenarioList <- CREATE scenario list
  FOR EACH scenario IN scenarios DO
    FOR EACH activationFlow FROM useCase DO
      activationScenario <- CREATE scenario FROM activationFlow steps
      newScenario <- COPY OF scenario
      INSERT activationScenario AT THE BEGINNING OF newScenario
      ADD newScenario TO combinedScenarioList
    ENDFOR
  ENDFOR
  REPLACE scenarios WITH combinedScenarioList IN useCase
  // 3. Combine preconditions
  scenarios <- useCase scenarios
  combinedScenarioList <- CREATE scenario list
  preconditions <- preconditions that are postconditions from other
                   use cases IN useCase
  FOR EACH precondition IN preconditions DO
    scenariosToCombine <- scenarios FROM the precondition's use case
                          CONTAINING precondition
    FOR EACH stc IN scenariosToCombine DO
      FOR EACH scenario IN scenarios DO
        newScenario <- COPY OF scenario
        INSERT stc AT THE BEGINNING OF newScenario
        ADD newScenario TO combinedScenarioList
      ENDFOR
    ENDFOR
  ENDFOR
  REPLACE scenarios WITH combinedScenarioList IN useCase
ENDFOR
END
```

Listagem 3 - Algoritmo em pseudocódigo da combinação de cenários

4.6.Geração de casos de teste semânticos (Etapa 5)

A geração de testes semânticos valorados e com oráculo para um formato independente de *framework* de testes é gerado com base na definição dos casos de uso e, principalmente, de suas regras de negócio (seção 4.3). Os critérios utilizados para esta geração são definidos a seguir.

4.6.1.Geração de valores para os testes

De acordo com Meyers (1979), o teste de software torna-se mais eficaz se os valores de teste são gerados baseados na análise das condições de mudança de valores, as chamadas "condições de contorno" ou "condições limite". Nesta abordagem, que analisa os valores limite, os casos de teste exploram as condições que dão o maior retorno (recompensa), usando valores **acima** e **abaixo** destes limites.

De acordo com o British Standard 7925-1 (1998), o teste de software torna-se mais eficaz se os valores são particionados ou divididos de alguma maneira. Em geral, os valores de teste de partição são gerados dividindo a faixa ao **meio**.

Além disto, é interessante a inclusão do valor **zero** (0), que permite testar casos de divisão por zero, bem como o uso de valores **aleatórios**, que podem fazer o software atingir condições imprevistas.

Assim, para a **geração de valores válidos**, foram definidos os seguintes critérios:

1. Valor mínimo;
2. Valor imediatamente posterior ao mínimo;
3. Valor máximo;
4. Valor imediatamente anterior ao máximo;
5. Valor intermediário (divisão no meio da faixa permitida);
6. Zero, se este estiver dentro da faixa permitida;
7. Valor aleatório, dentro da faixa permitida.

E para a **geração de valores inválidos**, os seguintes critérios foram definidos:

1. Valor imediatamente anterior ao mínimo;
2. Valor aleatório anterior ao mínimo;
3. Valor imediatamente posterior ao máximo;
4. Valor aleatório posterior ao máximo;
5. Formato de valor inválido.

Para valores do tipo `String`, o *valor mínimo*, o *valor máximo* e o *zero* são referentes ao comprimento do texto.

Para valores do tipo `Double`, é levada em conta a maior precisão adotada nos valores mínimo e máximo, de acordo com o número de casas decimais após a vírgula. Por exemplo, ao definir `1,253` como valor mínimo e `1,71` como valor máximo, a ferramenta infere que a precisão a ser usada é de `0,001` (três casas decimais).

Para valores em que há uma regra *Igual A* (Seção 4.3.1), o *valor mínimo* e o *valor máximo* são referentes à, respectivamente, o primeiro e o último valor da lista. E a geração de valores aleatórios fora da faixa é realizada gerando-se valores não presentes na lista.

A geração de *formato inválido* se refere à negação da expressão regular definida. Em geral, a geração de valores baseada em uma expressão regular (ER) não é trivial e muitas soluções fazem a transformação da ER para um **autômato finito**,[26] para decidir se um valor é considerado válido (ou não) para a mesma. O presente trabalho faz uso da solução criada por Moller (2010), que usa a representação simbólica baseada nos intervalo de caracteres Unicode[27] e se mostra rápida, comparada a outras soluções.[28]

4.6.2.Geração de oráculos

A geração dos oráculos é realizada somente para a geração de valores

[26] Um autômato finito é uma máquina de estados finitos limitada a reconhecer uma classe de linguagem específica, usualmente uma linguagem formal, como uma expressão regular.
[27] http://www.unicode.org/
[28] A análise comparativa das soluções está disponível em: http://tusker.org/regex/regex_benchmark.html

considerados **inválidos**, onde é esperado que o SST apresente uma mensagem criticando a invalidade do valor. Sua geração depende da definição de um passo (em um fluxo) que faça a verificação dos elementos, segundo a sintaxe definida na Seção 4.2.4.1.

Como a definição de uma regra de negócio especifica a mensagem esperada para quando o valor for inválido, o gerador do caso de teste, ao gerar um valor inválido para um elemento, já obtém a mensagem esperada para o elemento em caso de haver um oráculo. Havendo um oráculo, este receberá as mensagens referentes aos elementos que ele verifica.

É importante observar que, apesar dos oráculos só serem gerados explorando as regras de negócio definidas, o que ocorrerá apenas para a geração de valores inválidos, caso o caso de uso se comporte de forma diferente da expectativa definida por um de seus fluxos, a execução de seu teste funcional irá obter *falha* como resultado. Isto é verdade, por exemplo, para quando não for encontrado um elemento (ex.: *widget*) definido num passo. Ou seja, não é necessário haver oráculos que verifiquem estes casos, pois os *frameworks* de teste já costumam verificá-los.

4.6.3.Geração de casos de teste

Os casos de teste gerados visam explorar se o SST funciona como esperado, para o caso do fornecimento de valores considerados válidos, e se ele acusa os valores inválidos, quando fornecidos. A Tabela 9 apresenta os casos de teste elaborados:

Tabela 9 - Tipos de caso de teste elaborados

#	Descrição	Deve concluir?[29]	Qtd. Testes
1	Somente obrigatórios	Sim	1
2	Todos os obrigatórios exceto um	Não	1 por elemento editável obrigatório

[29] Se é esperado que o caso de uso tenha sua execução concluída gerando suas pós-condições.

3	Todos com valor/tamanho mínimo	Sim	1
4	Todos com valor/tamanho posterior ao mínimo	Sim	1
5	Todos com valor/tamanho máximo	Sim	1
6	Todos com valor/tamanho anterior ao máximo	Sim	1
7	Todos com o valor intermediário, dentro da faixa	Sim	1
8	Todos com zero, ou um valor aleatório dentro da faixa, se zero não for permitido	Sim	1
9	Todos com valores aleatórios dentro da faixa	Sim	1
10	Todos com valores aleatórios dentro da faixa, exceto um, com valor imediatamente anterior ao mínimo	Não	1 por elemento editável
11	Todos com valores aleatórios dentro da faixa, exceto um, com valor aleatório anterior ao mínimo	Não	1 por elemento editável
12	Todos com valores aleatórios dentro da faixa, exceto um, com valor imediatamente posterior ao máximo	Não	1 por elemento editável
13	Todos com valores aleatórios dentro da faixa, exceto um, com valor aleatório posterior ao máximo	Não	1 por elemento editável
14	Todos com formato permitido, exceto um	Não	1 por elemento com formato definido

Baseado nisto, podemos calcular a **quantidade mínima de casos de teste semânticos gerados por cenário** de caso de uso, QMTC, como:

$$QMTC = 8 + 4\mathrm{E} + \mathrm{O} + \mathrm{F}$$

Onde:

- *E* é o número de elementos editáveis;
- *O* é o número de elementos editáveis obrigatórios;
- *F* é o número de elementos editáveis com formato definido.

Logo, para um cenário bastante simples, que envolva apenas 5 campos, sendo 3 obrigatórios e 1 com formatação, teremos 8 + 3 + 4*5 + 1 = 32 casos de teste por cenário. E se este caso de uso tiver 3 cenários, poderá ter 96 casos de teste.

4.6.4.Valores fornecidos para outros casos de uso

Como discutido na Seção 4.5, quando é preciso que um caso de uso seja combinado com outros dos quais depende, são selecionados destes apenas os cenários que terminam com sucesso. Para que estes cenários possam funcionar como esperado, os valores fornecidos para os elementos de cada passo devem ser válidos. Desta forma, para estes elementos serão gerados **valores válidos aleatórios**, dentro da faixa permitida.

4.6.5.Cobertura

Através da consideração de **todas as regras de negócio** definidas, explorando seus valores limítrofes e outros (seção 4.6.1), espera-se obter uma boa taxa de cobertura para o caso de uso.

Como o número total de testes necessário para testar completamente um caso de uso não é conhecido, acredita-se que a cobertura de todos os seus cenários, aliada à cobertura de todas as suas regras de negócio, possam exercitar consideravelmente o SST, obtendo alta eficácia na descoberta de defeitos.

4.6.6.Formato dos casos de teste

A ferramenta construída gera casos de teste semânticos valorados e com oráculos (CTSVO), cuja sintaxe, na BNF, é apresentada na Listagem 4, a seguir.

```
<semantic-test-suite>           ::= <name> <software-name>
                                <creation-date-time>
                                <semantic-test-case>+
<semantic-test-case>            ::= <name> <use-case-name>
                                <scenario-name> <include-file>*
                                <semantic-test-method>+
<semantic-test-method>          ::= <name> <strategy-kind>
                                <expected-success>
                                <semantic-steps>+
<semantic-step>                 ::= <semantic-action-step>
                                | <semantic-oracle-step>
<semantic-action-step>          ::= <id> <step-id> <action-name>
                                <semantic-element>+
<semantic-oracle-step>          ::= <id> <step-id> <action-name>
                                <message>+
<semantic-element>              ::= <type> <internal-name> <name>
                                <value> <value-considered-valid>
                                <value-option>
<value-option>                  ::= 'MIN' | '   MAX' | 'ZERO'| 'MIDDLE'
                                | 'RIGHT_AFTER_MIN'
                                | 'RIGHT_BEFORE_MAX'
                                | 'RANDOM_INSIDE_RANGE'
                                | 'RIGHT_BEFORE_MIN'
                                | 'RIGHT_AFTER_MAX'
                                | 'RANDOM_BEFORE_MIN'
                                | 'RANDOM_AFTER_MAX'
                                | 'INVALID_FORMAT'
<name>                          ::= string
<software-name>                 ::= string
<creation-date-time>            ::= datetime
<use-case-name>                 ::= string
<scenario-name>                 ::= string
<include-file>                  ::= string
<strategy-kind>                 ::= string
<expected-success>              ::= boolean
<id>                            ::= long
<step-id>                       ::= long
<action-name>                   ::= string
<message>                       ::= string
<type>                          ::= string
<internal-name>                 ::= string
<value>                         ::= string
<value-considered-valid>        ::= boolean
```

Listagem 4 - Sintaxe dos casos de teste semânticos valorados

Os CTSVO são atualmente exportados para a *JavaScript Object Notation* (JSON) (CROCKFORD, 2006), um formato independente de *framework* de testes e linguagem de programação (apesar de ser um formato nativo da linguagem JavaScript). Este formato foi escolhido, ao invés da XML (WORLD WIDE WEB CONSORTIUM, 1998), por ser mais compacto, mais fácil de analisar gramaticalmente (realizar *parsing*), ser mais fácil de ler e editar por seres humanos e ter biblioteca de manipulação disponível em quase todas as linguagens de programação.

Um exemplo de arquivo JSON gerado a partir dos CTSVO pode ser visualizado na Seção 6.7.6.

4.6.7.Algoritmos

Por ser um processo *ad hoc*, variando conforme o tipo de teste desejado, seus algoritmos variam quase na mesma proporção. Nesse contexto, há algoritmos para a geração casos de teste, algoritmos para a geração de valores dependendo do tipo de dado e algoritmos para geração de valores conforme as regras de negócio.

Para efeito ilustrativo, serão apresentados dois algoritmos. O primeiro realiza a geração do conteúdo de um **método de teste semântico valorado para valores válidos**, permitindo parametrizar a *opção de valor válido* desejado (conforme as opções descritas na Seção 4.6.1). Sua ideia é a de transformar cada passo (de um cenário) que não for um *oráculo* nem uma *documentação* em um passo semântico que receberá um valor gerado conforme a opção parametrizada. Este algoritmo é exposto na Listagem 5, a seguir.

```
ALGORITHM GenerateSemanticMethodWithValidValues(
  IN scenario,
  IN validOption,
  IN justTheRequired
  ) RETURNING SemanticTestMethod
USING valueGenerator
BEGIN
  otherElementValues <- CREATE a map FOR element, object
  semanticStepList <- CREATE semantic step list
  FOR EACH step IN scenario DO
    IF NOT step is performable OR step is oracle THEN
      CONTINUE
    END
    semanticStep <- CREATE semantic step FROM step
    FOR EACH element IN step DO
      IF element is usecase THEN
        CONTINUE
      ENDIF
      semanticElement <- CREATE semantic element FROM element
      IF element is editable THEN
        IF NOT element is required AND justTheRequired is true THEN
          CONTINUE
        END
        value <- CALL generateValidValue( validOption, element, oth-
erElementValues ) FROM valueGenerator
        CALL setValue( value ) FROM semanticElement
        CALL setValueIsConsideredValid( true ) FROM semanticElement
        CALL setValueOption( validOption AS string ) FROM seman-
ticElement
      ENDIF
      ADD semanticElement IN semanticStep
    ENDFOR
    ADD semanticStep IN semanticStepList
  ENDFOR
  semanticMethod <- CREATE semanticMethod WITH semanticStepList
  RETURN semanticMethod
END
```

Listagem 5 - Algoritmo em pseudocódigo para geração de método de teste cujos elementos contêm valores válidos

O segundo algoritmo realiza a geração de testes semânticos valorados e com oráculo, construindo um método de teste para cada elemento editável obrigatório. O propósito do teste é **gerar valores aleatórios válidos para cada elemento, mas deixar de fornecer valor para um elemento obrigatório**. O oráculo, então, verificará se a ausência do valor para o elemento é notada, o que corresponde ao SST exibir a mensagem configurada nas regras de negócio. O algoritmo é apresentado na Listagem 6, a seguir.

```
ALGORITHM GenerateSemanticMethodListForEachRequiredElement(IN scenario, IN validOption)
  RETURNING semanticMethodList USING valueGenerator
BEGIN
  otherElementValues <- CREATE a map FOR element, object
  semanticMethodList <- CREATE semantic method list
  oracleMessageMap <- CREATE a map FOR element, string
  requiredElements <- required elements IN scenario
  FOR EACH requiredElement IN requiredElements DO
    semanticMethod <- CREATE semantic method WITH name FROM scenario
    semanticStepList <- CREATE semantic step list
    FOR EACH step IN scenario DO
      IF NOT step is performable THEN
        CONTINUE
      ENDIF
      IF step is oracle THEN
        semanticOracleStep <- CREATE oracle step FROM step
        messages <- CALL extractMessagesFromMappedElements( oracleMessageMap )
        CALL setMessages( messages ) FROM semanticOracleStep
        ADD semanticOracleStep IN semanticStepList
        CONTINUE
      END
      semanticActionStep <- CREATE semantic action step FROM step
      FOR EACH element IN step
        IF element is use case THEN
          CONTINUE
        ENDIF
        semanticElement <- CREATE semantic element FROM element
        IF element is editable THEN
          IF NOT element is required THEN
            CONTINUE
          ENDIF
          IF element EQUAL TO requiredElement THEN
            message <- message FROM required business rule IN element
            ADD element, message IN oracleMessageMap
            CONTINUE
          END
          value <- CALL generateValidValue( validOption, element,
                   otherElementValues ) FROM valueGenerator
          CALL setValue( value ) FROM semanticElement
          CALL setValueIsConsideredValid( true ) FROM semanticElement
          CALL setValueOption( validOption AS string ) FROM semanticElement
        ENDIF // is editable
        ADD semanticElement IN semanticActionStep
      ENDFOR // element IN step
      ADD semanticActionStep IN semanticStepList
    ENDFOR
    SET semanticStepList IN semanticMethod
    ADD semanticMethod IN semanticMethodList
  ENDFOR
  RETURN semanticMethodList
END
```

Listagem 6 - Algoritmo em pseudocódigo para geração de método de teste que verifica elementos obrigatórios

4.7.Transformação em código-fonte (Etapa 6)

O processo de transformação em código-fonte é realizado por **extensões** da ferramenta, capazes de transformar o arquivo JSON contendo os CTSVO para os *frameworks* de testes e de interface gráfica desejados. Na versão atual, foi construída uma extensão para os *frameworks* TestNG[30] e FEST[31] (ambos discutidos na seção 5.2.2), visando usar a linguagem Java para testar aplicações construídas com Swing.

4.7.1.Algoritmo

O processo de conversão realizado por cada extensão é um tanto *ad hoc*, pois depende da estrutura da linguagem e dos *frameworks* alvo. Apesar disto, a ideia geral da conversão pode ser descrita pelo algoritmo em pseudocódigo, apresentado na Listagem 7, a seguir.

```
READ the file containing the semantic test cases TO objects
FOR EACH semantic test case IN objects DO
  CREATE a text stream to receive the file content
  ADD include files FROM the used frameworks TO the stream
  ADD include files FROM the semantic test case TO the stream
CREATE a class declaration using the semantic test case name
ADD the class declaration TO the stream
  FOR EACH semantic test method IN the semantic test case DO
    CREATE the method declaration using the semantic test method
    ADD the method declaration TO the stream
    FOR EACH semantic step IN semantic test method DO
      CREATE an instrumented command line using the semantic step
      ADD the command line TO the stream
    ENDFOR
  ENDFOR
  SAVE the stream TO a file named using the class name
ENDFOR
```

Listagem 7 - Algoritmo em pseudocódigo para conversão de testes semânticos valorados em código-fonte

Dependendo do *framework* de testes alvo, pode ser necessário criar algum

[30] http://testng.org
[31] http://fest.easytesting.org/

arquivo de configuração, possivelmente contendo quais arquivos de teste foram gerados, para que ele possa realizar a execução dos testes corretamente.

O SST deverá ser executado pelo código de teste, que irá começar a testá-lo, simulando a interação de um usuário. A forma como esta execução é realizada dependerá do *framework* de testes utilizado.

4.7.2.Instrumentação do código-fonte gerado

Para facilitar o rastreamento das falhas ou erros no código-fonte de testes gerado, a extensão construída realizou sua instrumentação, indicando, com comentários de linha, o passo semântico correspondente à linha de código de teste. Esta instrumentação será utilizada na Etapa 8, para pré-análise dos resultados (seção 4.9).

4.8.Execução do código-fonte (Etapa 7)

O código-fonte é executado por uma **extensão** da ferramenta (na versão corrente, a mesma usada para transformar os CTSVO), através de linhas de comando configuradas pelo usuário, na ferramenta. Estas linhas de comando podem incluir a chamada a um compilador, ligador (*linker*), interpretador, navegador *web*, ou qualquer outra aplicação ou arquivo de script que dispare a execução dos testes.

Durante a execução do código-fonte, o *framework* de testes utilizado gera um *log* de execução, que será lido e analisado pela extensão na próxima etapa do processo.

4.9.Coleta, pré-análise e transformação dos resultados de execução (Etapa 8)

O *log* de execução gerado pelo *framework* de testes será lido pela **extensão**, após a execução dos testes, para:

1. Analisar os testes que falharam ou obtiveram erro, investigando:

    1.1. A **mensagem de exceção** gerada, para detectar o tipo de erro

ocorrido – que varia conforme os *frameworks* gerados – e fornecer uma informação adicional, independente de *framework*, para a análise (posterior) da **ferramenta**;

2.1. O **rastro da pilha de execução**, para detectar o arquivo e a linha de código onde a exceção ocorreu e, então, ler a **instrumentação** do código gerada na de transformação em código-fonte (seção 4.7.2), de forma obter a identificação do passo semântico correspondente ao código. Esta identificação do passo semântico será usada posteriormente pela **ferramenta**, em sua análise, para identificar, além do teste semântico, o passo, fluxo, caso de uso e cenário correspondentes.

2. Converter o *log* para um formato independente de *framework* e linguagem de programação, esperado pela ferramenta;

Esta pré-análise deve ser feita pela **extensão**, e não pela **ferramenta**, pois ela quem conhece os detalhes da instrumentação realizada e dos *frameworks* de teste utilizados. O processo realizado por esta etapa é *ad hoc*.

4.9.1.Sintaxe

A sintaxe (BNF) dos resultados de execução dos testes esperado pela ferramenta é dada pela `<test-execution-report>`, definida a seguir, na Listagem 8.

```
<test-execution-report>  ::= <creation> <tool-name>
                             <target-language>
                             <target-gui-frameworks>
                             <target-test-frameworks>
                             <original-test-result-file>
                             <test-suite-execution>+
<test-suite-execution>   ::= <test-execution> <name>
                             <test-case-execution>+
<test-execution>         ::= <total-tests> <total-passed>
                             <total-skipped> <total-failures>
                             <total-errors> <total-unknown>
                             <time-in-millis>
<test-case-execution>    ::= <class-name> <test-method-execution>+
<test-method-execution>  ::= <name> <is-for-configuration>
                             <time-in-millis> <status> <exception-class>
                             <exception-message> <stack-trace>
                             <erroneous-file> <erroneous-file-number>
                             <erroneous-line-of-code>
                             <erroneous-semantic-step-id>
<status>                 ::= 'PASS' | 'FAIL' | 'ERROR' | 'SKIP' | 'UNKNOWN'
<creation>                       ::= date
<tool-name>                      ::= string
<target-language>                ::= string
<target-gui-frameworks>          ::= string
<target-test-frameworks>         ::= string
<original-test-result-file>      ::= string
<name>                           ::= string
<total-tests>                    ::= integer
<total-passed>                   ::= integer
<total-skipped>                  ::= integer
<total-failures>                 ::= integer
<total-errors>                   ::= integer
<total-unknown>                  ::= integer
<time-in-millis>                 ::= long
<class-name>                     ::= string
<is-for-configuration>           ::= boolean
<exception-class>                ::= string
<exception-message>              ::= string
<stack-trace>                    ::= string
<erroneous-file>                 ::= string
<erroneous-file-number>          ::= integer
<erroneous-line-of-code>         ::= string
<erroneous-semantic-step-id>     ::= long
```

Listagem 8 - Sintaxe dos resultados da execução dos testes

Atualmente o arquivo com o resultado de execução dos testes possui formato JSON, pelos mesmos motivos expostos na seção 4.6.6.

4.10.Coleta, análise e apresentação dos resultados (Etapa 9)

Por fim, a ferramenta realiza a leitura do arquivo com os resultados da execução dos testes e procura analisá-los para rastrear as causas de cada problema possivelmente encontrado. Nesta etapa, o resultado da execução é confrontado com a especificação do software, visando identificar possíveis problemas.