4 Metodologia e ferramenta desenvolvida

Nesse capítulo descreveremos o processo proposto desde a modelagem de uma funcionalidade GUI como Rede de Petri até a execução da suíte de teste gerada.

Apresenta-se a ferramenta externa MISTA [14], que será utilizada para a modelagem gráfica da Rede de Petri. Detalhamos a sintaxe e a semântica utilizadas na Rede de Petri de forma a representar os elementos e propriedades da GUI, bem como a interação entre eles. Em seguida apresentam-se as simulações da rede, utilizadas para observar o comportamento do modelo e os cenários de interação, os quais, nesse contexto, representam os casos de teste. Para auxiliar na explicação utiliza-se um exemplo genérico e simples de Rede de Petri.

Após a introdução sobre a metodologia, apresenta-se a aplicação do processo em uma funcionalidade real do software de estudo de caso deste trabalho.

Por fim, é dada uma visão de como a ferramenta guiftG, desenvolvida para este trabalho, interpreta os logs de execução e o arquivo xml com a estrutura da rede para então gerar a suíte de teste em C++.

No último tópico, falamos sobre a execução das suítes de teste no SUT, realizada por uma ferramenta externa.

4.1 Modelagem da Rede de Petri

A modelagem da RP deve ser realizada com uma ferramenta gráfica para que se tenha o benefício do formalismo visual. Como isso, foi realizada uma pesquisa à procura de uma ferramenta gráfica disponível e que desse suporte à RP de alto nível. A ferramenta selecionada foi a MISTA - Model-based Integration and System Test Automation.

Uma vez que temos as interações possíveis da GUI apresentadas e supondo que os requisitos sobre o seu comportamento também estão disponíveis, podemos começar a construir a Rede de Petri desta funcionalidade.

A ferramenta MISTA oferece um suporte bem intuitivo à construção da Rede de Petri. A rede vai sendo desenhada tal como num software de desenho livre, onde se escolhem os itens e anotações que serão inseridos no gráfico, num menu superior. A Figura 10 é um *screenshot* mais amplo da funcionalidade, que

mostra o menu superior com os itens que se pode selecionar para criar passo a passo a Rede de Petri. Além dos itens gráficos, pode-se ainda clicar duas vezes sobre uma transição, um lugar ou um arco, para que sejam inseridas anotações específicas, como nomes e condições de guarda.

As próximas figuras de Rede de Petri deste trabalho conterão apenas a rede em si, sem mostrar toda a ferramenta, para não ocupar espaço desnecessário no documento.

4.1.1 Construção do modelo da GUI

Para iniciar a modelagem é preciso definir primeiramente:

1) Quais as ações possíveis de ocorrer na GUI e devem fazer parte dos casos de teste.
2) Quais *widgets* da GUI e quais das suas propriedades serão verificadas. Por exemplo, uma combo XYZCombobox precisa estar desativada (propriedade active=false) em determinado instante, de acordo com a especificação.

Na Rede de Petri, a transição representa um evento ou um processamento, logo as transições, representadas graficamente por retângulos cheios, correspondem ao item 1) acima. Os lugares na RP definem algum tipo de objeto, que, quando receberem *tokens* valorados (o valor deve ser de um tipo que faça sentido para a propriedade a ser verificada), estarão em determinado estado e passam a conter um oráculo (serão "verificáveis"). Nesse caso, os itens em 2) serão representados pelos lugares, os quais graficamente são círculos.

Veja o exemplo genérico da Figura 10 abaixo, referente a uma interface gráfica com dois campos de texto para notas de provas P1 e P2, um botão que calcula a média aritmética e um campo onde será exibido o resultado:

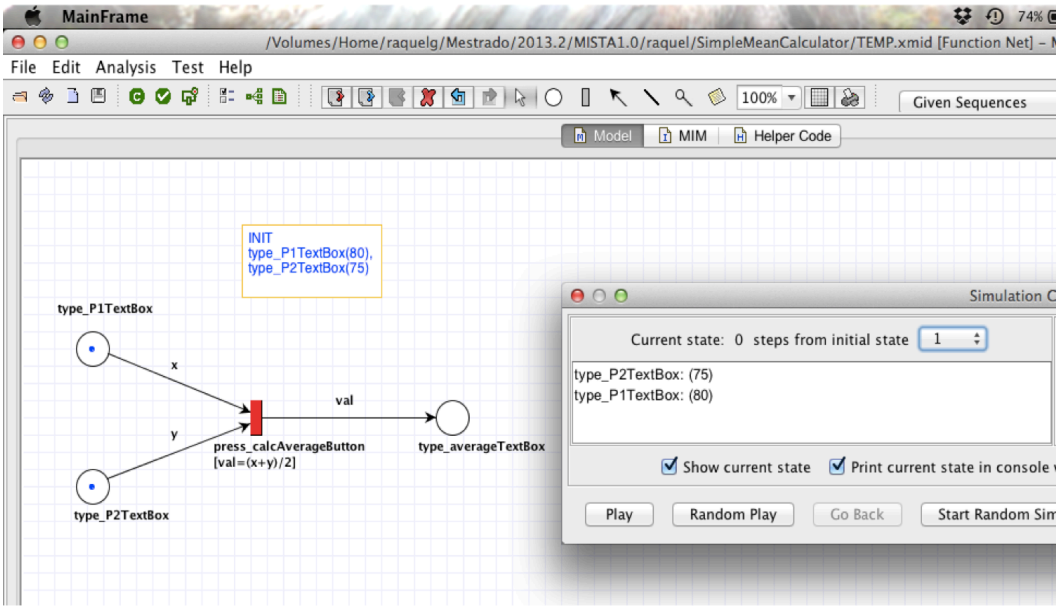

Figura 10 – Rede de Petri representando uma interface que calcula a média entre duas notas

Como pode ser observado, a sintaxe adotada tanto para lugar como para transição é: **propriedade_nomeWidget**. A transição é uma ação sobre uma propriedade do *widget*. O valor vinculado à ação é o valor que será aplicado na propriedade e esse valor é definido por um *token* que entra por um arco de entrada da transição e que deve ter o mesmo nome do prefixo da propriedade. Na rede acima, a transição press_calcAverageButton não tem nenhum arco de entrada de nome "press", nesse caso um valor default é atribuído à transição pela ferramenta guiftG. O valor default para press é 1 (=true), o que significa que o botão é pressionado.

A transição produz *tokens,* via arcos de saída. O valor do *token* que transitará por cada arco de saída é definido ou por um arco de entrada com o mesmo rótulo (variável) do arco de saída, ou na *guard condition* da transição. A *guard condition* é um local opcional para se definir condições em função dos arcos de entrada ou valores, na forma de expressão aritmética, a variáveis em arcos de saída. [14]

Um arco de entrada numa transição é dito uma pré-condição da transição e essa condição é satisfeita quando existe (pelo menos) um *token* no lugar associado. A pré-condição da transição do exemplo acima na Figura 10 é: type_P1TextBox(x) && type_P2TextBox(y)*. Se houvesse alguma condição definida na *guard condition*, ela também faria parte da pré-condição da transição. Um arco de saída numa transição é dito uma pós-condição e ela é satisfeita através da produção de um *token* no lugar associado, após o disparo [14]**.** A pós-condição da transição do exem-

plo acima na Figura 10 é o lugar type_averageTextBox(val)*, onde val tem seu valor definido pela expressão que calcula a média, a qual representa o processamento realizado nesta transição. Quando um *token* de valor val chegar ao lugar type_averageTextBox(val), este *token* será um oráculo. Se na transição houvesse mais dois lugares de saída, por exemplo PlaceX(a) e PlaceY(b), sua pós-condição seria: type_averageTextBox(val) && PlaceX (a) && PlaceY(b), análogo à pré-condição.

A ordem dos acontecimentos no disparo de uma transição é a seguinte: Quando uma transição dispara, ela primeiro, necessariamente consome os *tokens* dos lugares de entrada, removendo uma unidade de *token* de cada um. Imediatamente após, ela produz um *token* em cada lugar de saída. Após o disparo, a rede apresenta um novo estado e cada *token* presente nela é um oráculo de teste, o qual será uma verificação no caso de teste.

Neste trabalho, os tipos de propriedades e *widgets* que podem ser usados na Rede de Petri, pois são interpretados pela ferramenta guiftG, são os da Figura 11 abaixo:

* A semântica de type_P1TextBox(x), type_P2TextBox(x) e type_averageTextBox, denota que este lugares armazenam o valor inserido nos campos P1TextBox, P2TextBox e averageTextBox. O prefixo "type*"* indica a propriedade a ser verificada, apesar de o nome não expressar exatamente o nome de uma propriedade. Poderia ser "content" ou "value", por exemplo. "type" foi escolhido como forma de unificar com o prefixo "type" utilizado em transições, que significaria "digitar".

Propriedades no contexto da transição – Ação
type
check
select
press
remove
rename

Widgets
nnnnTextBox
nnnnComboBox
nnnnLabel
nnnnSlider
nnnnSpin
nnnnRadio
nnnnToggle
nnnnButton
nnnnWindow

Propriedades no contexto do lugar – Verificação
type
active
check
select
press
open
close
hide
numElements

Figura 11 – Sintaxes de *widgets* e propriedades a serem usadas na Rede de Petri

As ações **remove** e **rename** não são ações generalizadas para *widgets* de interface, tal como as outras. Elas são específicas do sistema apresentado e foram criadas para permitir mais tipos de interação. Voltaremos a falar sobre elas no capítulo com o exemplo real.

Após a definição dos lugares e transições, deve-se fazer as ligações entre lugares e transições com arcos direcionais ou bidirecionais, nomeá-los e definir as condições e expressões nas *guard conditions*, quando necessário. Os arcos direcionais possuem uma seta numa das extremidades e denotam que o *token* navega naquela direção. Já os arcos bidirecionais, não possuem seta em nenhuma das extremidades e representam dois arcos direcionais em direções contrárias e de mesmo nome. Isso significa que quando existe um arco bidirecional, um mesmo *token* é consumido e produzido pela transição, ou seja, quando a transição dispara ela "mantém" o *token* no lugar de entrada.

Por fim, deve-se definir a marcação inicial, a qual indica o estado inicial da rede e denota a configuração inicial da interface gráfica. A forma de definir a marcação inicial na ferramenta gráfica é através de um campo de texto com a palavra INIT, seguida pelos nomes dos lugares onde se deseja especificar *tokens* iniciais. Na Figura 10, a rede está no seu estado inicial e a transição press_calcAverageButton, que representa o clique no botão de calcular média, está habilitada, pois há *tokens* em todas as suas entradas e não há nenhuma condição a ser satisfeita na *guard condition* em função dos arcos de entrada. Na *guard condition* existe a definição do valor do *token* de saída, através da expressão que calcula a média entre dois valores, representando a rotina executada naquele evento.

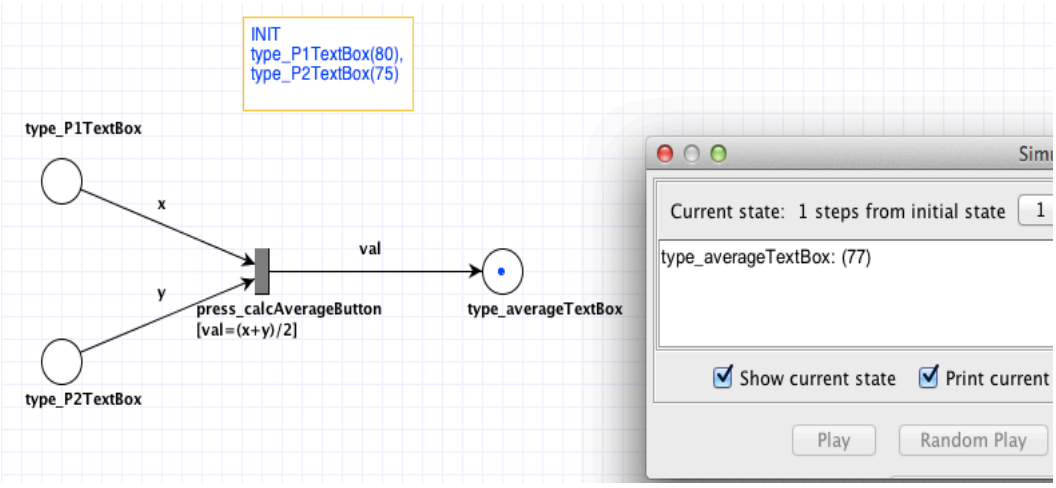

Figura 12 – Rede de Petri após o disparo da única transição

Na Figura 12, a simulação da rede disparou a transição, consumindo um *token* de cada entrada e produzindo um *token* na saída, cujo valor foi especificado

na *guard condition*. No caso de teste gerado, após a ação de clique no botão calcAverage, acontece a verificação em cima do campo de texto averageTextBox, o qual deve ter o valor 77, dado que os *tokens* entrantes foram 80 e 75.

Agora a transição não se encontra mais habilitada porque não há mais *tokens* em suas entradas.

Vamos incrementar esta funcionalidade, adicionando duas transições e um lugar, tal como na Figura 13. Na nova rede temos as transições type_P1TextBox e type_P2TextBox, que representam as ações de usuário que inserem valores nos campos de texto. O lugar __init, com o nome precedido por dois *underscores,* é uma convenção deste trabalho para indicar que este lugar não é um *widget* da GUI e portanto, quando nele houver *tokens*, não haverá uma verificação no caso de teste gerado pela ferramenta guiftG. Estes lugares são ignorados na hora da construção dos casos de teste. O lugar __init pode armazenar diversos *tokens* que serão os dados de teste para a calculadora de média.

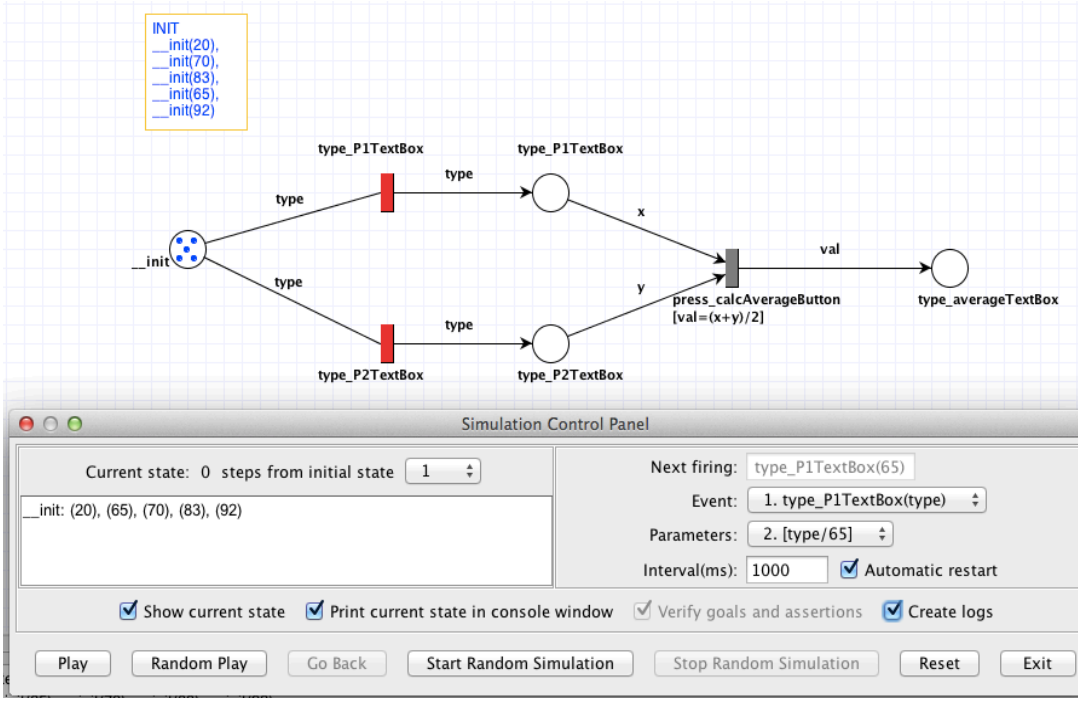

Figura 13a - Rede de Petri do exemplo da média, em seu estado inicial

Na Figura 13a a nova rede está no estado inicial. Tanto os arcos de entrada, quanto os arcos de saída das transições type_P1TextBox e type_P2TextBox, que digitam valores nos campos de notas, têm o mesmo nome type. Com isso, o valor que a transição consumir é o mesmo que ela irá produzir no lugar de saída, não

sendo necessário definir a variável do arco de saída na condição de guarda. Uma outra observação é que os dois arcos que saem de __init são bidirecionais. Como dito, uma transição ligada a um arco bidirecional, consome o *token* de um lugar e produz outro de mesmo valor, consequentemente quando esta transição dispara, ela "mantém" o *token* no lugar. Isso é útil nessa rede para que o testador possa executar várias vezes o mesmo caso de teste com outros dados de teste. Além disso, torna a rede mais realística porque um usuário pode realizar diversas vezes o evento de digitação de valores nos campos de texto e pressionar o botão de calcular no momento que ele quiser.

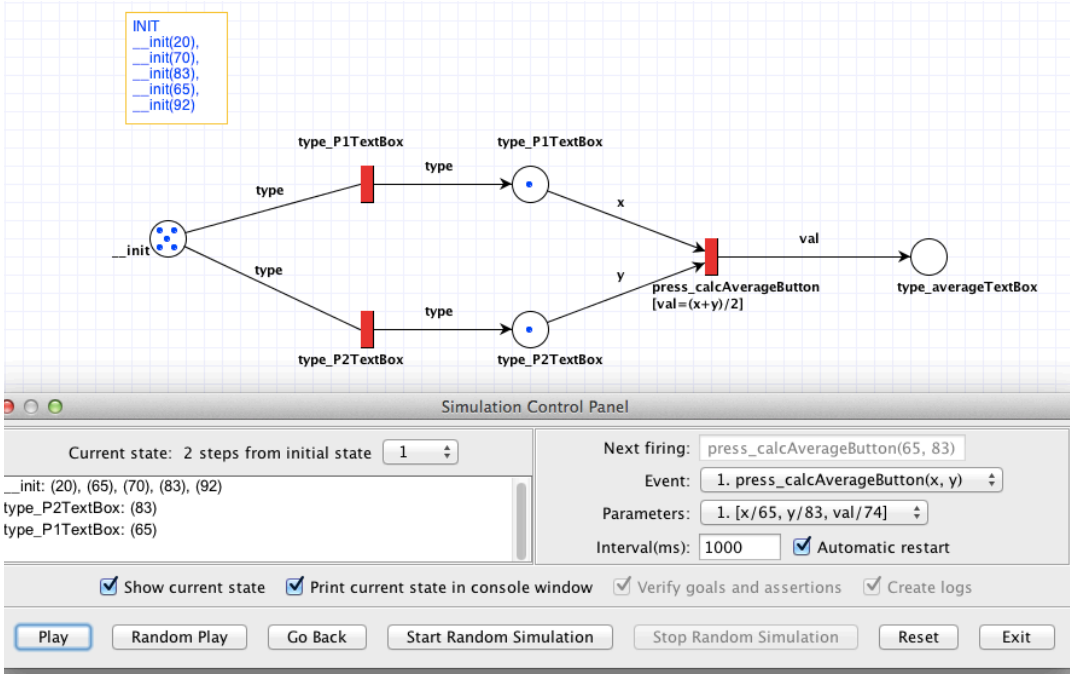

Figura 13b

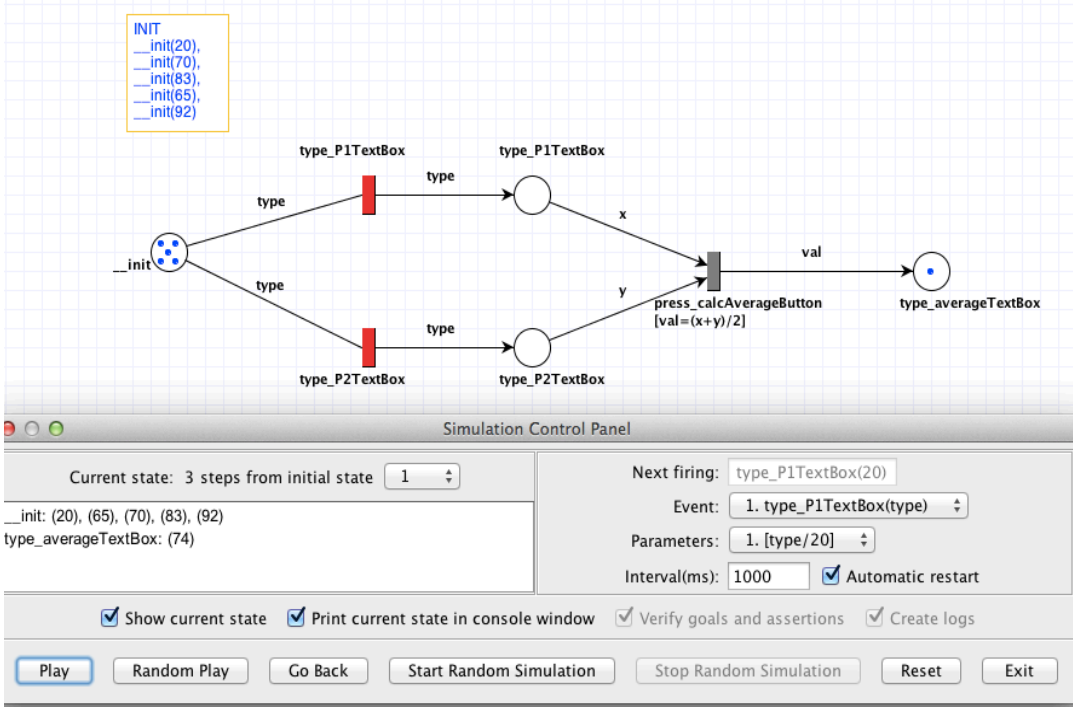

Figura 13c

Na Figura 13b as duas primeiras transições já dispararam e na Figura 13c o botão de calcular a média foi pressionado, removendo os dados dos campos de texto e produzindo um *token* com o valor calculado no campo de média.

4.1.2 Simulação

A simulação ou execução é uma função disponível em Redes de Petri, logo cada ferramenta de modelagem de RP deve fornecer esta aplicação. A execução da rede é feita passo a passo, onde um passo é um disparo de transição. A janela de execução contém o botão "Play", o qual o testador pressiona cada vez que deseja executar o próximo passo, podendo logo em seguida verificar o estado da rede. Antes de apertar "Play", o testador pode escolher qual transição das habilitadas ele quer que dispare e quais os *tokens* (valores) serão consumidos, daqueles que estão disponíveis nos lugares de entrada. [14]

As Figura 14 abaixo, assim como as Figuras 13a, 13b e 13c, são exemplos da janela na qual é realizada a simulação. O campo de cor branca à esquerda da janela mostra o estado corrente da rede. Isto significa que cada lugar que contém um ou mais *tokens,* aparecerá nesta lista à esquerda, com o valores dos *tokens* entre parêntesis.

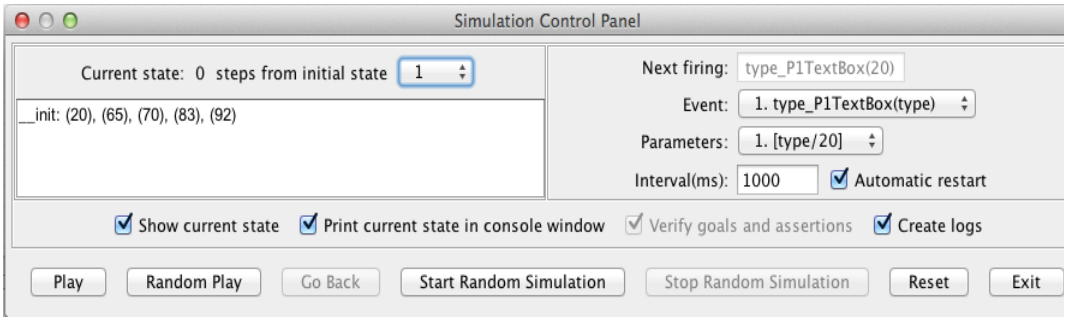

Figura 14 – Janela para a simulação da RP da ferramenta MISTA

A ferramenta de modelagem gráfica e simulação MISTA gera um log para cada execução realizada, que é igual a um caminho percorrido pela rede. Um caminho é uma sequência de eventos (disparos de transição) a partir de uma dada marcação inicial, e que termina, ou quando é atingido um estado final (aquele onde não há mais transições habilitadas), ou quando o testador interrompe a execução.

Cada log gerado representa um caso de teste. Para gerar outro caso de teste, o testador inicia uma nova execução, a qual deve ser diferente, ou na sequência de transições ou na marcação inicial selecionada. Após ter executado todos os fluxos desejados, têm-se um conjunto de logs de execução. Estes serão parte da entrada na ferramenta que gera o arquivo da suíte de teste.

Logo abaixo está o log da única execução da rede acima. Nele constam as três transições disparadas, na ordem em que ocorreram, com seus respectivos parâmetros de entrada (valor dos *tokens* consumidos no momento em que dispararam).

```
SEQUENCE (INIT 0) type_P1TextBox [type/65],
type_P2TextBox[type/83], press_calcAverageButton[x/65, y/83]
```

Na Figura 14 pode-se observar as opções e os itens disponibilizados na janela de simulação. Os botões “Play”, “Random Play” e “Start Random Simulation”; A opção “Create logs”; A ComboBox acima onde se escolhe a marcação inicial a ser utilizada para aquela simulação; O campo à esquerda, onde são exibidos todos os lugares com *tokens* e seus valores a cada instante; E à esquerda, as Combos Event e Parameters, onde são escolhidos, a transição que será disparada naquele passo e os parâmetros de entrada.

O tipo de execução mencionada anteriormente é manual, no entanto, a ferramenta MISTA fornece um tipo de simulação automática e randômica, a qual é

mais interessante do ponto de vista da eficiência do processo, bem como da eficácia da suíte gerada, como será discutido mais adiante.

A simulação automática da Rede de Petri é disponibilizada pela ferramenta através do botão "Start Random Simulation". Este botão fornece a mesma característica de execução mencionada acima, mas o usuário não precisa pressionar o botão "Play" para cada disparo de transição. Também não é necessário escolher qual transição irá disparar nem os seus parâmetros, já que estes serão escolhidos de forma automática e aleatória pela ferramenta. Ao pressionar "Start Random Simulation", a ferramenta escolhe uma das marcações iniciais disponíveis e simula a rede até encontrar um estado final ou até que o número máximo (configurável) de disparos seja atingido. Deixando a opção "Automatic restart" selecionada, a simulação automática continua. Isso significa que ela não cria somente um caso de teste, ou seja, quando a primeira simulação termina, uma outra inicia automaticamente e assim por diante, gerando outros casos de teste. A simulação automática será encerrada quando o usuário quiser e para isso deve pressionar o botão "Stop Random Simulation".

Em testes aleatórios, o conjunto de entradas do teste precisa ser extenso [27], para garantir que boa parte das alternativas de execução sejam cobertas. Na Rede de Petri, para que se possa obter uma cobertura suficiente das combinações possíveis, o número de casos de teste e a extensão deles precisa ser grande, o que implica deixar a simulação rodando por um tempo tal que gere um determinado número de testes e um determinado tamanho de sequência de execução. A priori, não sabemos os números ideais de casos de teste e a extensão deles. Seria necessário um experimento que variasse esses números, medindo a eficácia a cada variação, até chegar a um número aparentemente ótimo. Neste trabalho, realizamos os experimentos com estes valores fixos. O motivo para isso e mais detalhes sobre o experimento serão comentados no capítulo 6.

O caráter automático de simulação permite que, após a modelagem da rede, a geração de todas as sequências de eventos que irão compor suíte sejam geradas automaticamente. Ao término da simulação, basta que o usuário entre com os arquivos de log gerados e com o arquivo estrutural da Rede de Petri, na ferramenta guiftG, que irá gerar a suíte de teste executável. Essa etapa é apresentada na seção 4.2 deste capítulo.

O caráter randômico de simulação está ligado às escolhas aleatórias: Da transição a ser disparada a cada passo, dos parâmetros da transição a ser disparada e da marcação inicial de cada nova simulação.

Na abordagem aleatória de testes de software, o domínio das entradas para o teste é identificado e as entradas são independentemente escolhidas. [27-28] No tipo de teste aqui apresentado, as entradas são as transições disponíveis a cada passo e as marcações iniciais disponíveis para iniciar um caso de teste. Em testes randômicos, como a entrada de dados é aleatória e em cada execução o teste pode gerar saídas diferentes, o oráculo não pode ser definido previamente. Uma das possiblidades é que o oráculo seja baseado numa especificação formal, tipicamente uma assertiva executável, ou no uso de funções inversas para regerar os dados que foram fornecidos. A especificação provê o necessário para a verificação das saídas. No nosso caso, a Rede de Petri é a especificação, que após cada passo contém o estado concreto da GUI, fornecendo então o oráculo do teste.

De acordo com Hamlet em [27]**,** a maior parte dos critérios de teste surgiram com a ideia de "cobertura". Os testes sistemáticos são muitas vezes intuitivamente mais atraentes, porque visam garantir que cobrirão todos (ou uma porcentagem alta) os pontos que podem conter erros. O teste aleatório, por outro lado, não tem indicativo de nenhum tipo de cobertura, exceto o que for ditado pelo acaso. No entanto, ainda segundo Hamlet, mesmo sob suposições favoráveis aos testes sistemáticos, estudos mostraram que eles não são melhores em revelar problemas do que os testes aleatórios [27-28]. Testes sistemáticos seriam mais subjetivos do que aparentam, porque em muitos casos há várias maneiras de satisfazer um critério e a escolha é feita manualmente. Na prática, o testador pode não ter informação ou habilidade suficiente para fazer uma boa escolha. Além disso, nem sempre o tipo de cobertura selecionada é útil em expor as falhas. Isso se aplica bem ao caso de GUIs, que por apresentar grande complexidade na relação entre estados resultantes de eventos e o contexto anterior, faz com que os critérios de cobertura tradicionais, como *all-transitions* e *all-states*, não sejam tão confiáveis.

4.1.3 Exemplo de funcionalidade real

Nesta seção apresentamos uma funcionalidade do sistema usado como prova de conceito deste trabalho. A interface gráfica apresenta uma complexidade baixa em termos de número de operações (eventos) possíveis e número de elemen-

tos gráficos, o que a fez ser uma boa candidata a exemplo. A Figura 15 é o *screenshot* da funcionalidade, chamada Operação sobre Horizontes. No capítulo 6, serão apresentadas funcionalidades mais complexas também modeladas para este trabalho.

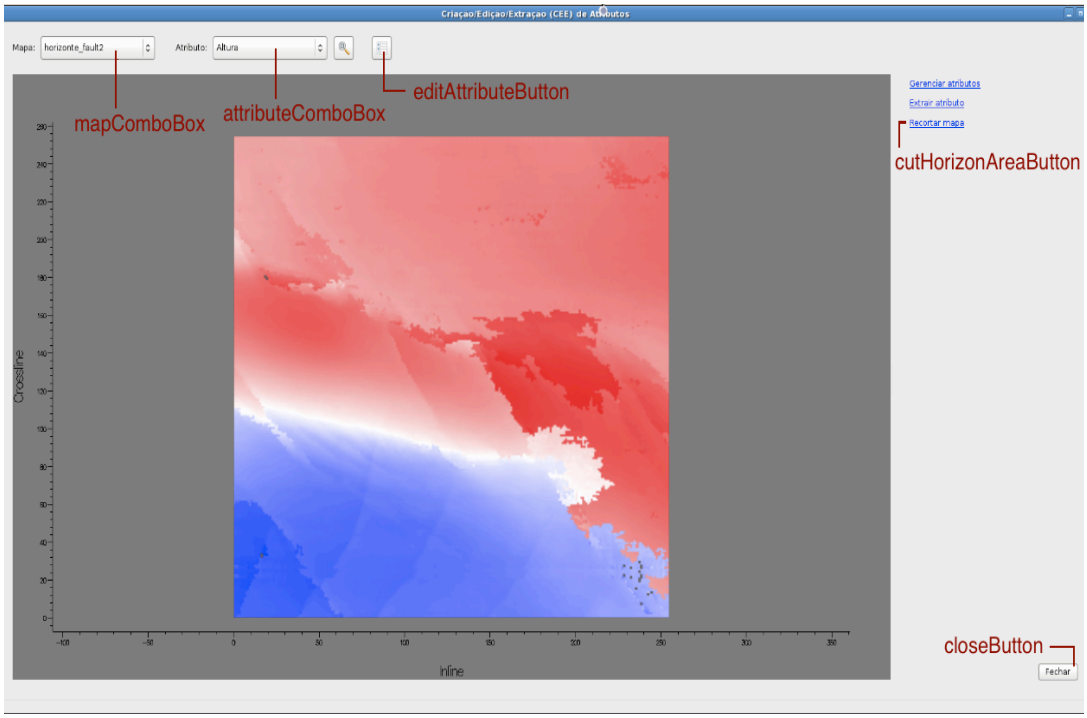

Figura 15 – *Screenshot* de uma das funcionalidades modeladas como Rede de Petri

A janela é aberta ao clicar-se em um dado sísmico do tipo Horizonte, que está carregado em memória e sendo exibido na janela principal do software, onde existe o canvas 3D (Figura 16).

Um horizonte sísmico é uma interface entre duas camadas sub-superfície de sedimentos com diferentes características físicas. Ele é definido como uma série de reflexões contínuas de ondas sísmicas de intensidades similares, geradas no processo de aquisição de dados geológicos. A ocorrência de um horizonte num dado sísmico 3D (volume sísmico) significa que ali ocorreram eventos (picos ou vales de amplitudes sísmicas) consistentes ao longo do dado. Um arquivo de horizonte é composto essencialmente por um conjunto de pontos de coordenadas x e y e um valor no eixo Z (ou “altura”) para cada ponto.

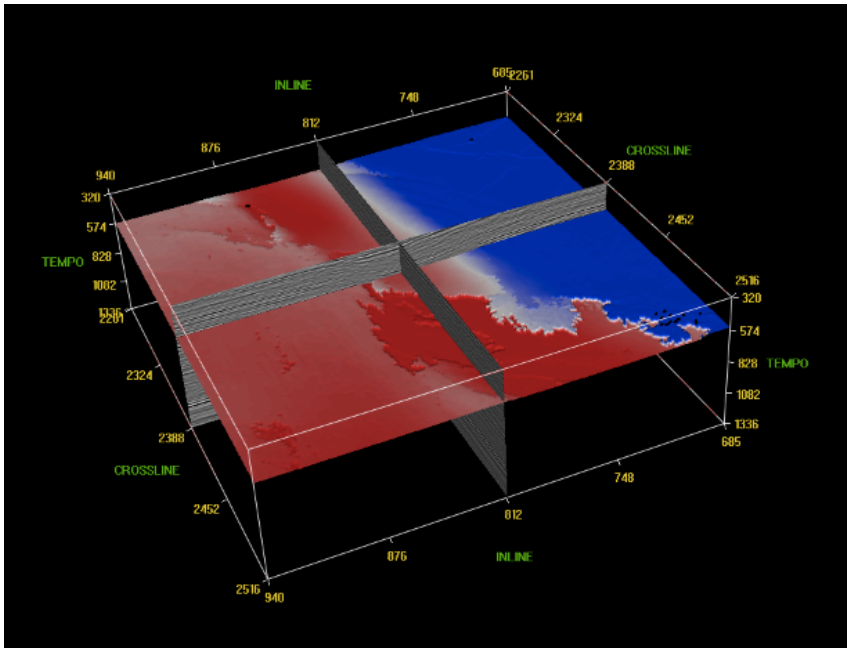

Figura 16 – Canvas 3D do software de estudo

A funcionalidade Operações sobre Horizonte (Figura 15) permite que o geofísico edite o horizonte de várias formas, tais como: Remover ou renomear um atributo, criar um atributo numa região específica ou no horizonte inteiro, editar os valores de uma região, recortar uma região, dentre outras. Ao mesmo tempo em que manipula o dado, o usuário tem uma visão de topo do horizonte e que permite acompanhar o que está acontecendo com ele.

Todas as operações possíveis listadas acima estão disponíveis nos links que aparecem na porção superior direita da Figura 15. Ao clicar em um deles, abre-se uma aba com a interface gráfica da sub-funcionalidade. O exemplo que mostraremos aqui é relativo à interface gráfica primária da funcionalidade Operações sobre Horizontes, que é exatamente o que mostra a Figura 15. Nela, os elementos de interface disponíveis são:

mapComboBox - ComboBox com horizontes para edição/visualização.

attributeComboBox - ComboBox com atributos do horizonte visualizado.

editAttributeButton - Botão “Remover ou Renomear atributo”.

cutHorizonAreaButton - Link para a sub-funcionalidade Recorte de mapa.

closeButton - Botão “Fechar”, que encerra a funcionalidade e fecha a janela.

Há ainda interações de mouse no canvas 2D, como *move*, *zoom* e *pan*, que alteram a visualização do horizonte. Mas estes eventos fogem à primeira intenção

deste trabalho, pois requerem outro tipo de verificação, que não propriedades de *widgets*.

Na funcionalidade de exemplo temos o seguinte caso de teste: Ao clicar em um dos links que abrem as sub-funcionalidades (por exemplo, o link "Recortar mapa" (cutHorizonAreaButton)), a Combobox de seleção de horizonte (mapComboBox) deve ser desabilitada, porque foi especificado que não se poderia trocar o horizonte corrente estando alguma sub-funcionalidade já aberta. Se a funcionalidade de recorte for fechada (cutCloseButton), mapComboBox volta a ficar habilitada. Pode-se ainda verificar se a aba de Recorte realmente abriu ou fechou quando houve clique no link cutHorizonAreaButton ou no botão cutCloseButton. Portanto, vamos começar mostrando esse caso de teste, que é apenas um pedaço da rede maior. A Figura 17 mostra essa sub-rede.

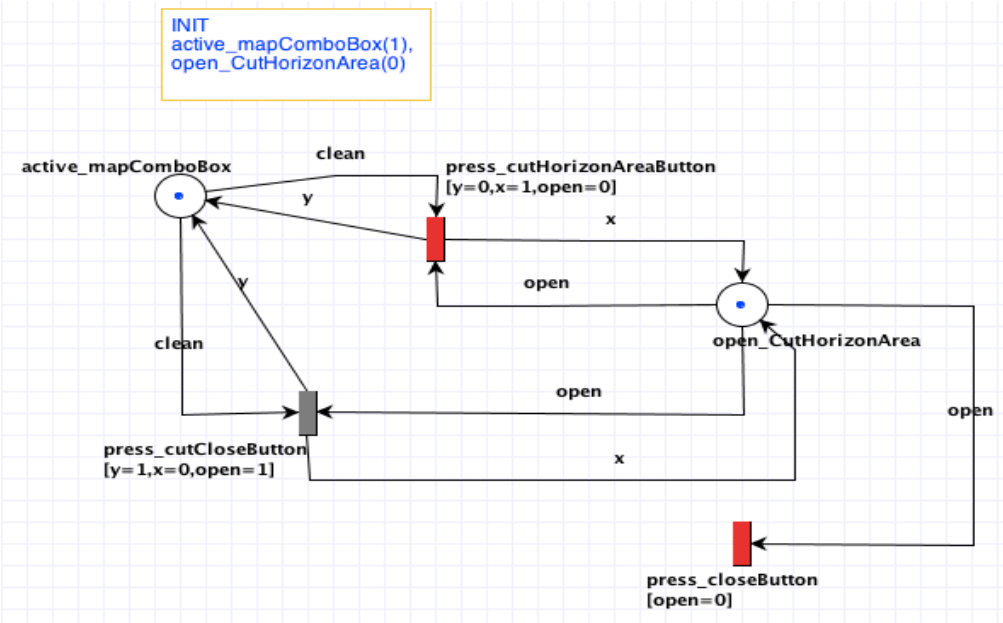

Figura 17 – Parte da Rede de Petri da funcionalidade Operações sobre Horizontes

Os dois *tokens* inicialmente definidos em active_mapComboBox e open_CutHorizonArea têm valores 1 e 0 respectivamente, significando que a combo de horizontes está inicialmente habilitada e a aba da funcionalidade de Recorte fechada (open=0). Para propriedades "booleanas" como active, open, close e hide, adotou-se valores de *tokens* iguais a 1 ou 0, denotando *true* ou *false*. Neste exemplo, ao longo da simulação, as transições (exceto a press_closeButton, que encerra a funcionalidade) estão sempre consumindo e produzindo um *token* nos lugares

aos quais elas estão ligadas. Ao produzir um novo *token,* dão a ele um valor que faça sentido para o próximo estado em determinado *widget*. Neste exemplo, a "conservação" dos *tokens* nos lugares acontece porque, como já mencionado, queremos verificar o estado dos *widgets* após a transição.

A Figura 18 abaixo é a Rede de Petri completa da funcionalidade Operações sobre Horizontes, já em estado inicial de simulação.

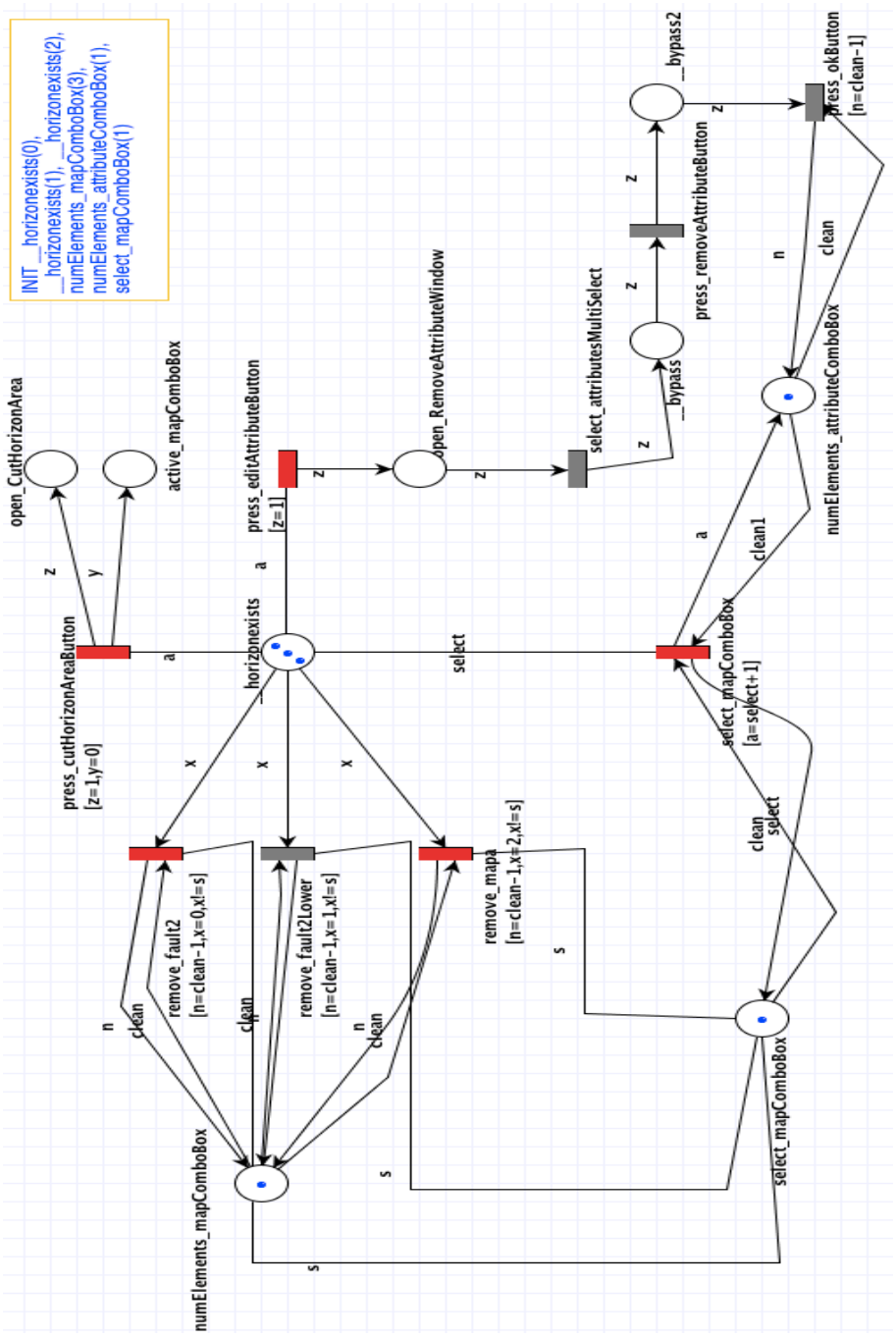

Figura 18 – Rede de Petri da funcionalidade Operações sobre Horizontes

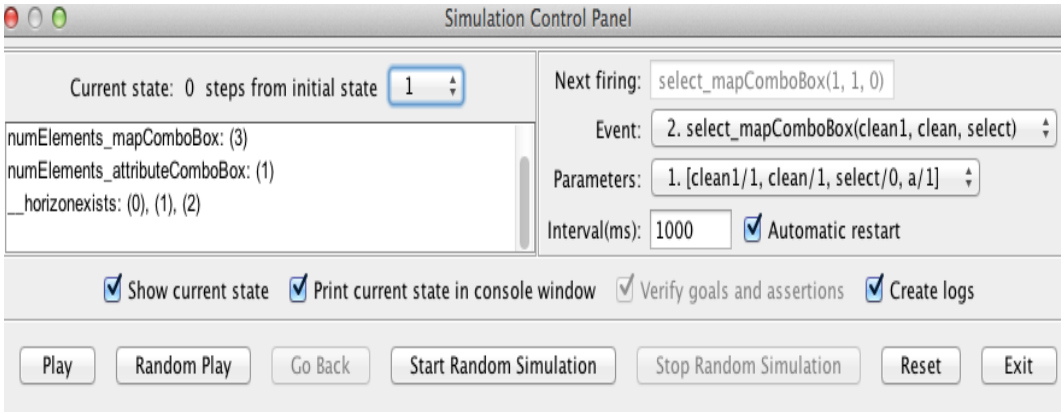

Figura 19 – Janela de simulação da Rede de Petri acima, ilustrando um dado instante

Como já mencionado, as ações **remove** e **rename** não são ações generalizadas para *widgets* de interface, tal como as outras. São específicas do sistema e foram criadas para permitir mais tipos de interação. Elas representam ações que o usuário pode realizar de fora da janela em teste. São ações onde se remove ou renomeia algum objeto do projeto corrente e esse evento implica em uma alteração na janela sob teste, como a redução no número de elementos de uma Combo ou a alteração de uma Label, dentre outros.

Na Figura 18 a marcação inicial está definida na diretiva INIT. Nela os seguintes *tokens* iniciais estão definidos:

Três *tokens* de valores 0, 1 e 2 no lugar central __horizonexists. Os valores representam índices para os três horizontes que existirão no projeto onde será realizado o teste. Os valores poderiam ser strings com os nomes dos horizontes, mas por questão de modelagem foram escolhidos índices.

Um *token* de valor 3 no lugar numElements_mapComboBox, representando que a combobox de horizontes começa com o número de elementos igual a três, já que é a quantidade de horizontes disponíveis.

Um *token* de valor 1 no lugar numElements_attributeComboBox, representando o número de elementos iniciais nesta combobox. A combo de atributos tem inicialmente um elemento porque o horizonte selecionado inicialmente na combo de horizontes possui um atributo.

Um *token* de valor 1 no lugar select_mapComboBox, representando que a combo de horizontes estará selecionada inicialmente no índice 1.

Dada essa marcação inicial, tem-se que algumas transições estão habilitadas para disparo, o que é análogo às interações possíveis de um usuário assim que a janela é aberta.

As transições à esquerda removem determinado horizonte do projeto. Essa ação é realizada no software por fora da janela corrente e implica em mudanças no estado de alguns elementos da janela sob teste. Por exemplo, se a transição remove_fault2 disparar, ela consome o *token* relativo ao índice daquele horizonte na combo, diminuindo a quantidade de horizontes disponíveis em __horizonexists. O disparo também atualiza o valor do *token* em numElements_mapComboBox, para uma unidade a menos.

Das transições à esquerda, a do meio está desabilitada inicialmente porque o horizonte em questão está selecionado na combo. De acordo com a especificação, não é possível remover esse horizonte do projeto se ele estiver em uso por outra janela. Esta mesma transição, pode ser habilitada se a transição select_mapComboBox (mais abaixo na rede), for disparada com outro valor de índice de horizonte.

A transição press_editAttributeButton habilitada à direita representa o clique no botão da porção superior da funcionalidade, que quando pressionado, abre uma outra janela menor onde se pode remover ou renomear atributos do horizonte corrente. Então este caminho mais à direita da rede denota um fluxo executado neste bloco da interface gráfica.

A seguir estão os quatro logs gerados na simulação desta funcionalidade. Alguns são sequências curtas e outras longas.

```
SEQUENCE (INIT 0) press_cutHorizonAreaButton [a/0]
```

```
SEQUENCE (INIT 0) select_mapComboBox [clean1/1, clean/1, se-
lect/0], select_mapComboBox [clean1/1, clean/0, select/2],
select_mapComboBox [clean1/3, clean/2, select/1],
remove_mapa [s/1, clean/3, x/2],
press_cutHorizonAreaButton [a/0]
```

```
SEQUENCE (INIT 0) select_mapComboBox [clean1/1, clean/1, se-
lect/0], select_mapComboBox [clean1/1, clean/0, select/1],
remove_mapa [s/1, clean/3, x/2],
select_mapComboBox [clean1/2, clean/1, select/1],
press_editAttributeButton [a/0], select_attributesMultiSelect
[z/1], press_removeAttributeButton [z/1], press_okButton [z/1,
clean/2], select_mapComboBox [clean1/1, clean/1, select/0]
```

```
SEQUENCE (INIT 0) remove_mapa [s/1, clean/3, x/2], se-
lect_mapComboBox [clean1/1, clean/1, select/1],
select_mapComboBox [clean1/2, clean/1, select/0],
remove_fault2Lower [s/0, clean/2, x/1]
```

4.1.4 Configurações extras de modelagem

Verificação de caminhos inválidos e erros no sistema.

Existem casos onde queremos modelar opções de interação na interface que não estão de acordo com a especificação, representando caminhos ou entradas inválidas no sistema. Estes casos podem exercitar mensagens de erro ou mesmo derrubar o programa. Devemos testar os casos de mal uso e entradas inválidas para verificarmos o comportamento quando isso acontece.

Uma característica marcante nas implementações de GUIs no V3O2 e em muitos outros sistemas é a proteção a entradas incorretas de dados na camada lógica através da restrição dos eventos que podem acontecer na GUI. Por exemplo, se em determinado momento, um botão não deveria ser pressionado ou uma combo não deveria ter sua seleção alterada, estes *widgets* estarão desabilitados, evitando que eventos não esperados ou incorretos tenham que ser tratados na camada lógica.

Como estas restrições sobre o comportamento dos *widgets* é uma característica das GUIs deste trabalho, as RPs modeladas as consideram. Dessa forma, algumas transições possuem pré-condições evitando eventos inválidos, ou seja, muitas vezes pode-se perceber que os eventos só tornam-se habilitados na RP quando são os permitidos/válidos de acordo com a especificação. No entanto, isso não impede que caminhos inválidos sejam testados, por dois motivos:

1) Não é necessário deixar uma transição habilitada (quando esta não deveria estar) para garantirmos o teste de um caminho inválido.

Basta que o *widget* responsável por disparar aquela transição seja verificado quanto a sua disponibilidade, pois quando ele estiver desabilitado, é garantido que o evento não ocorrerá. Então, um evento que não pode acontecer em determinado instante (um evento inválido), pode ser representado por uma transição desabilitada e conter um lugar como pré-condição, cujo *token* verifica se o *widget* está disponível. No teste, se o *widget* estiver habilitado, quando não deveria estar, o teste falha e já indica o problema. (Figura 20)

2) Pode-se modelar normalmente o comportamento que gera o erro.

Para testar entradas inválidas que não estão restringidas na interface gráfica e que exercitam algum tipo de erro no sistema, uma solução é modelar o comportamento que gera o erro. Para isso, cria-se uma transição do evento que causa o erro. A saída da transição pode estar ligada a dois tipos de lugares: Um lugar comum, que verifica algum *widget* indicador de erro, como uma *Label* em vermelho que aparece quando um campo de texto está vazio, ou um botão "Aplicar" que deve estar desabilitado enquanto aquele campo não está preenchido. Ou um lugar especial com nome "Error", que quando contiver um *token*, indicará a verificação sobre a variável global que indica a ocorrência de um erro. Um exemplo disso nas funcionalidades modeladas neste trabalho é quando um usuário tenta remover do projeto, um objeto, como um horizonte ou volume sísmico. Quando estas funcionalidades estão operando, o objeto corrente não pode ser removido. Se for, o sistema acusa um erro, apresenta uma mensagem e volta a operar. Podemos então modelar uma transição que permite a remoção de um objeto. Após o disparo, ela produz um *token* no lugar de saída "Error", que representa o oráculo do teste. (Figura 21)

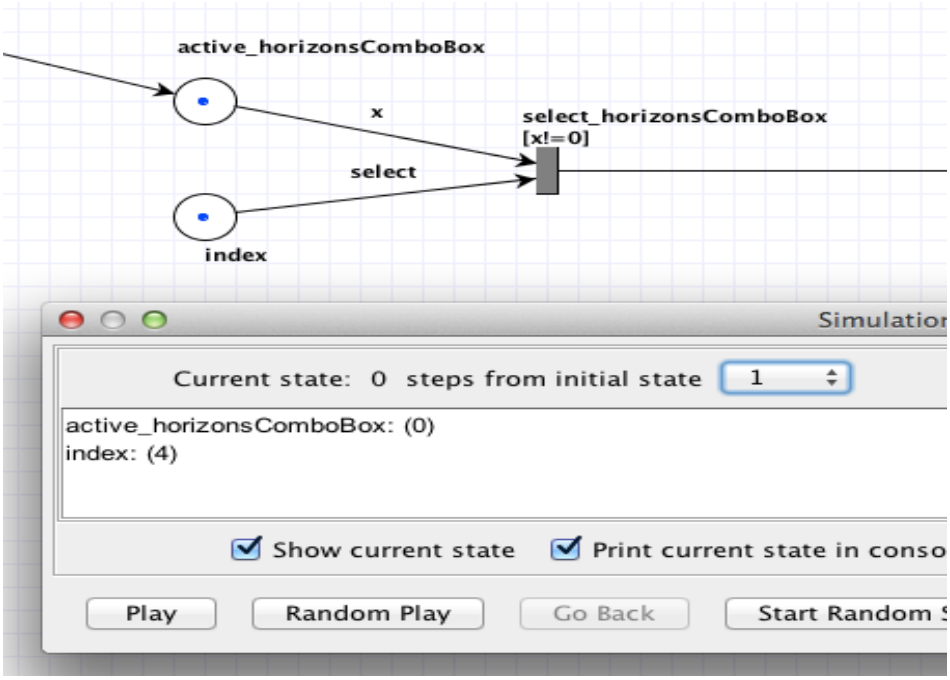

Figura 20 – Evento indisponível por haver um *widget* desabilitado

Na figura 20, a transição **select_horizonsComboBox** (desabilitada)**,** que representa um evento de seleção de um índice na combo **horizonsComboBox**, têm como pré-condição para o seu disparo as seguintes três condições: Deve conter um *token* no lugar **index**, denotando que há um índice a ser escolhido; Deve haver um *token* no lugar **active_horizonsComboBox**; E o valor desse *token* deve ser diferente de zero (zero representa falso). Ou seja, a combo **horizonsComboBox** deve ter a propriedade active igual a 1, para que o evento de seleção na combo ocorra. Na figura, a transição está desabilitada e o evento, que seria inválido, não ocorrerá. Isso representa corretamente o especificado para a interface, de que não haverá clique de seleção na combo. O foco do teste nesta situação é garantir que a combo esteja realmente desabilitada, impossibilitando que o evento inválido aconteça nesse momento.

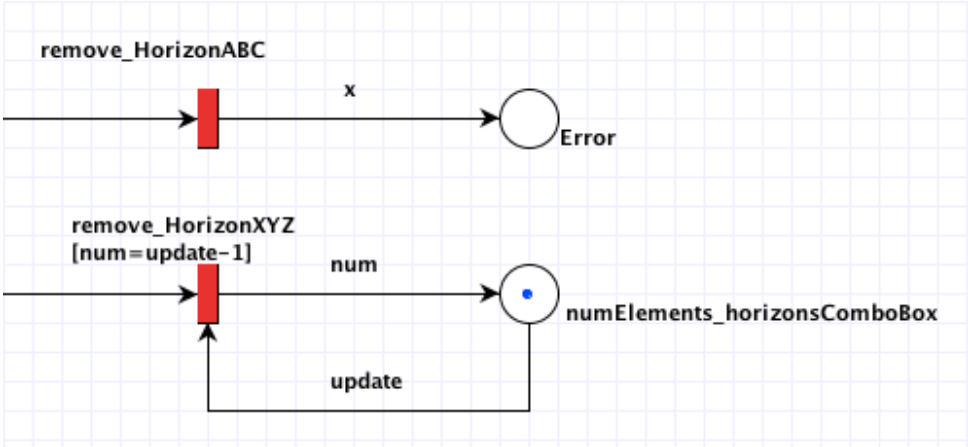

Figura 21 – Exemplo de um evento que gera um estado de erro.

Na figura 21, uma transição remove o horizonte HorizonABC e a outra remove o HorizonXYZ. Ambas podem disparar, ou seja, estes eventos podem acontecer de forma não-determinística. No entanto, a transição mais acima causa um erro, porque o horizonte está sendo usado no momento, enquanto a segunda não causa nenhum erro e apenas atualiza o número de horizontes na combo.

Verificações com thread em andamento.

Quando um evento dispara uma thread do programa, o código de teste gerado para as verificações posteriores não deve ser imediatamente executado pelo script de teste, a não ser que o intuito do testador seja realmente testar o estado de

alguns *widgets* enquanto a thread está em andamento. Por exemplo, o botão que dispara uma thread deve estar desabilitado enquanto ela estiver em andamento, para que não seja pressionado novamente. Por outro lado, o objetivo do testador pode ser o de realizar as verificações somente após o término daquela execução.

Essas duas considerações exigem que o testador indique na transição se ela dispara alguma thread. A regra adotada para esse tratamento é a seguinte: Esse tipo de transição deve receber uma anotação na condição de guarda com a palavra **thread_checknow** ou **thread_wait**. A primeira indica que as verificações posteriores a esse evento devem ser realizadas enquanto a thread está em andamento. A segunda indica que é preciso esperar a thread terminar para dar sequencia às verificações. Com essa indicação na transição, o código gerado irá considerar a espera ou não do término da execução da thread, para realizar as verificações seguintes. A Figura 22 mostra um exemplo desse uso:

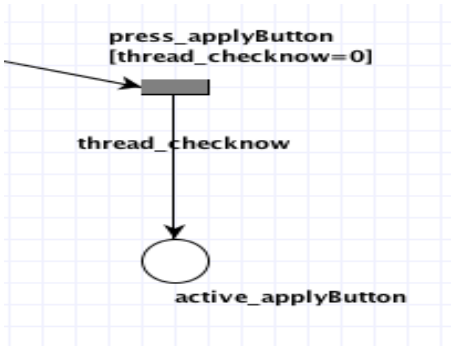

Figura 22 – Transição com anotação na condição de guarda, indicando que o evento dispara uma thread e que as verificações devem ocorrer com a thread em andamento.

4.2 Geração da suíte de teste – Ferramenta guiftG

Para que o processo estudado e proposto neste trabalho agregasse valor, uma das necessidades era que a suíte de teste conceitualmente formada através da Rede de Petri, fosse transformada automaticamente para um formato concreto e executável por uma ferramenta de teste automático. Para isso, foi desenvolvida a ferramenta guiftG – GUI Functional Test Generator. O desenvolvimento foi feito na linguagem C++ e com o toolkit de interface QT.

A Figura 23 é a única interface gráfica desta ferramenta.

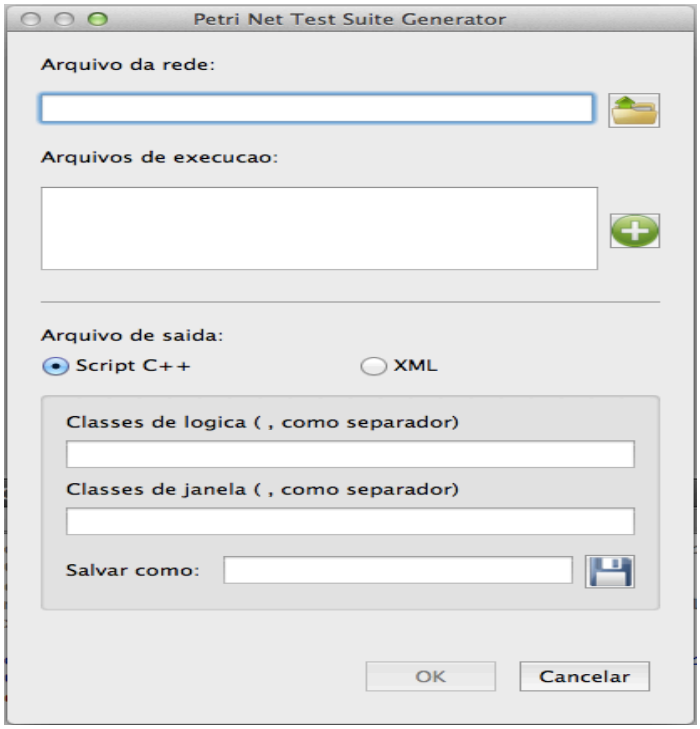

Figura 23 – Interface gráfica da ferramenta guiftG

No primeiro campo informa-se o arquivo XML que a ferramenta MISTA fornece com a estrutura da Rede de Petri. No segundo campo, devem ser fornecidos os arquivos de log de execução também fornecidos pela MISTA. A seguir o usuário escolhe em qual formato a suíte de teste deve ser gerada, o que será discutido nas seções 4.2.1 e 4.2.2, e os nomes das classes envolvidas no teste, necessárias para o script executável.

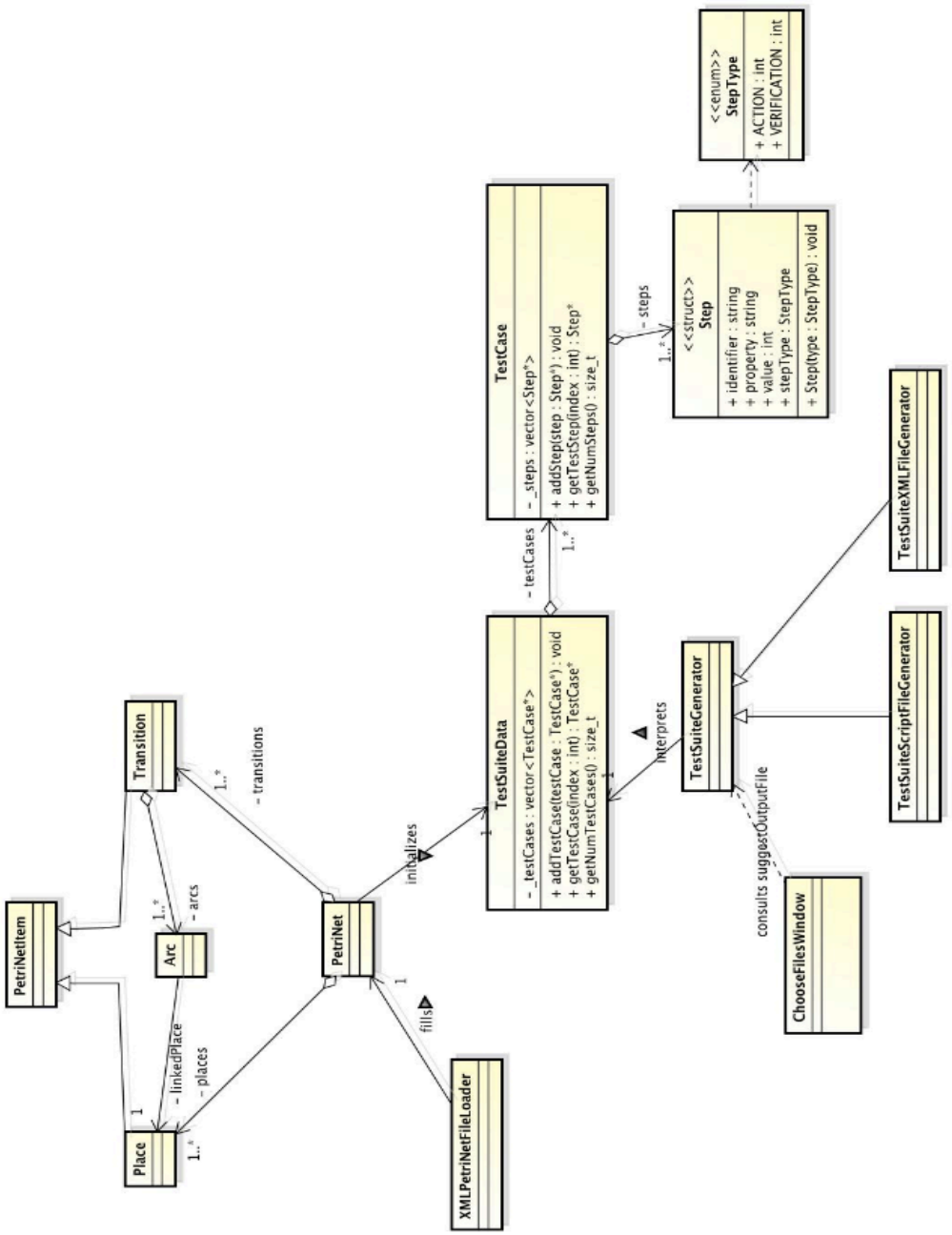

Figura 24 – Diagrama de classes da ferramenta desenvolvida guiftG.

Ao clicar no botão OK da ferramenta, a classe XMLPetriNetFileLoader interpreta os arquivos xml e log, preenchendo um objeto do tipo PetriNet com a estrutura da rede, suas ligações, guard conditions, marcações iniciais e fluxos de execução fornecidos pelo log. A partir de então, temos a rede pronta para executar em memória, exatamente como a ferramenta MISTA outrora realizou. A classe PetriNet, através da função execute(), realiza essa execução dadas as regras de funcionamento da Rede de Petri. Ao mesmo tempo em que executa todo o fluxo, a guiftG preenche a estrutura com os casos de teste gerados. Para isso existe a classe

TestCase, que contém Step's, e a classe TestSuiteData que contém um conjunto de TestCase's.

Portanto, ao término da função execute() o programa tem um objeto TestSuiteData com todos os casos de teste, compostos por sequências de passos do tipo ação e do tipo oráculo. A partir de então, basta que haja um módulo responsável por traduzir esta suíte em memória para um formato desejado.

4.2.1 Geração da suíte de teste "genéricos"

Visando a generalidade da automatização implementada pela ferramenta guiftG, a primeira versão desta ferramenta gerou suítes de teste num formato genérico. Esse formato é baseado em XML e foi definido no trabalho de Wanderley e Staa [29].

Para gerar a suíte de teste neste padrão o usuário seleciona a opção XML na seção "Arquivo de saída" na interface da guiftG. A classe TestSuiteXMLFileGenerator realiza essa tarefa e o arquivo gerado pode ser então interpretado por outra ferramenta que implemente a geração de scripts de testes executáveis em qualquer linguagem.

Outra forma de facilitar o uso desta solução para outros sistemas cuja linguagem de scripts utilizada seja diferente é fornecer uma API para a comunicação com a guiftG, mediando o acesso à estrutura TestSuiteData e permitindo então a geração de scripts no formato desejado.

```
<TestCase>
    <Oracles>
        <VerifyComponent>
            <Identifier>p3TextBox</Identifier>
            <Verify>
                <Property>type</Property>
                <Value>""</Value>
            </Verify>
        </VerifyComponent>
        <VerifyComponent>
            <Identifier>calcButton</Identifier>
            <Verify>
                <Property>active</Property>
                <Value>0</Value>
            </Verify>
        </VerifyComponent>
    </Oracles>
    <Actions>
        <Action>
            <Identifier>p3TextBox</Identifier>
            <Type>type</Type>
            <Value>8.0</Value>
        </Action>
    </Actions>
    <Oracles>
        <VerifyComponent>
            <Identifier>calcButton</Identifier>
            <Verify>
                <Property>active</Property>
                <Value>1</Value>
            </Verify>
        </VerifyComponent>
        <VerifyComponent>
            <Identifier>p1TextBox</Identifier>
            <Verify>
                <Property>type</Property>
                <Value>7.0</Value>
            </Verify>
        </VerifyComponent>
        <VerifyComponent>
            <Identifier>p2TextBox</Identifier>
            <Verify>
```

Figura 25 – Pedaço de arquivo XML de uma suíte de teste

4.2.2 Geração da suíte especializada para uma ferramenta de execução de teste automatizada

A linguagem de script de teste para o SUT neste trabalho é C++ e o script de teste em si deve ter uma organização especificada pela ferramenta de teste utilizada, o GoogleTest.

Para gerar a suíte de teste no formato executável pelo GoogleTest o usuário deve selecionar a opção Script C++ na seção “Arquivo de saída” na interface da guiftG. A classe TestSuiteScriptFileGenerator realiza essa tarefa e a ferramenta apresenta como saída um arquivo de extensão cpp, que nada mais é do que a classe de teste (script) a ser inserida no projeto do SUT e que passará a fazer parte do conjunto de testes de interface existentes.

Na prática, a suíte gerada precisa sofrer ajustes manuais, porque existe uma complexidade envolvida em gerações automáticas de código a partir de modelo em geral. Quanto menos padronizado for o código, em termos de classes da interface

gráfica, parâmetros recebidos, nomes de métodos, dentre outros, mais difícil será gerar um código sincronizado com as tarefas a serem executadas. No entanto, o trabalho manual requerido para ajustar as classes de teste geradas foram sempre menores que uma hora.

```
1  #include "gtest/gtest.h"
2  #include "HorizonOperationManagerPresenter.h"
3  #include "HorizonOperationsWindow.h"
4  #include "tests/TestsUtil.h"
5  #include "gtk/gtk.h"
6  #include "v3o2.h"
7  #include "projeto.h"
8
9  class HorizonOperationManagerPresenterTest : public HorizonOperationManagerPresenter
10 {
11 };
12 class HorizonOperationsWindowTest : public HorizonOperationsWindow
13 {
14 };
15
16 class HorizonOperationsUITest : public ::testing::Test
17 {
18 public:
19 HorizonOperationsUITest(){}
20 virtual ~HorizonOperationsUITest(){}
21 };
```

```
55 TEST_F(HorizonOperationsUITest,testcase1)
56 {
57 V3O2* v3o2 = V3O2::instancia();
58 TestsUtil::run_gtk_queue();
59 v3o2->abrirSessao("/path_to_a_project/project.v3o2");
60 TestsUtil::run_gtk_queue();
61 Projeto* project = v3o2->projeto();
62
63 HorizonOperationManagerPresenter* presenter = new HorizonOperationManagerPresenter();
64 presenter->show();
65 TestsUtil::run_gtk_queue();
66
67 //Para ter acesso a janela e seus widgets
68 HorizonOperationManagerPresenterTest* presenterTest = dynamic_cast<HorizonOperationManagerPresenterTest*>(pres
69 HorizonOperationsWindowTest* windowTest = dynamic_cast<HorizonOperationsWindowTest*>(presenterTest->_window);
70
71 windowTest->_mapComboBox.selectItem(0);
72 TestsUtil::run_gtk_queue();
73
74 EXPECT_EQ(0, (int)windowTest->_mapComboBox.getSelectedIndex());
75
76 EXPECT_EQ(1, (int)windowTest->_attributeComboBox.getNumberOfElements());
77
78 EXPECT_EQ(3, (int)windowTest->_mapComboBox.getNumberOfElements());
```

```
79
80  windowTest->_mapComboBox.selectItem(1);
81  TestsUtil::run_gtk_queue();
82
83  EXPECT_EQ(1, (int)windowTest->_mapComboBox.getSelectedIndex());
84
85  EXPECT_EQ(2, (int)windowTest->_attributeComboBox.getNumberOfElements());
86
87  EXPECT_EQ(3, (int)windowTest->_mapComboBox.getNumberOfElements());
88
89  Objeto* object = project->findObject("mapa");
90  if (object != 0)
91  {
92  CmdRemoverObj removeCommand(object, project, object->getClass());
93  removeCommand.Executar();
94  }
95  TestsUtil::run_gtk_queue();
96
97  EXPECT_EQ(1, (int)windowTest->_mapComboBox.getSelectedIndex());
98
99  EXPECT_EQ(2, (int)windowTest->_mapComboBox.getNumberOfElements());
100
101 EXPECT_EQ(2, (int)windowTest->_attributeComboBox.getNumberOfElements());
102
103 windowTest->_cutHorizonAreaButton.clicked.trigger();
104 TestsUtil::run_gtk_queue();
105
106 EXPECT_TRUE(presenterTest->_window != 0);
107
108 EXPECT_EQ(1, (int)windowTest->_mapComboBox.getSelectedIndex());
109
110 v3o2->destroyProject();
111 TestsUtil::run_gtk_queue();
112 }
```

Figura 26 – Pedaço da classe de teste gerada: Início do arquivo e primeiro caso de teste.

4.3 Execução dos testes no SUT (Software Under Test)

A execução dos testes unitários e funcionais de interface gráfica no software V3O2 é realizado pela ferramenta GoogleTest, ou gtest. Ela provê uma biblioteca que é agregada ao executável do software.

O teste para uma classe pode ser criado num arquivo .cpp, com um cabeçalho inicial onde é declarada a classe de teste, com construtor, destrutor, setup, teardown e o que mais for necessário. O restante do arquivo é um conjunto de macros TEST_F(TestClassName, testCaseName), onde cada uma é um caso de teste do modulo em questão. Na main do sistema é inserido um código que roda todos os testes existentes quando em modo teste. Há também uma opção de filtro do gtest onde se pode executar somente um teste específico ou um caso de teste específico.

Os testes unitários e funcionais já existentes do software V3O2 são rodados duas vezes por dia em um servidor de integração contínua. Os testes gerados como fruto deste trabalho já passaram a fazer parte desta integração. No médio e no longo prazo, espera-se que estes testes encontrem problemas quando houver manutenções nessas funcionalidades.