1 Introdução

1.1 Testes de Software

Na etapa de testes de um software podem ser reveladas falhas que, em seguida, devem ser diagnosticadas a fim de identificar os defeitos que as causaram. Estes defeitos então serão corrigidos, antes que provoquem falhas nas mãos do usuário final. Deve-se frisar que é igualmente fundamental a realização de inspeções, visando especificações, modelos e código, durante a fase de desenvolvimento e antes dos testes. Estas técnicas aumentam a produtividade e a confiabilidade do software porque, como já reconhecido, quanto mais cedo os defeitos forem identificados, menor o custo para corrigi-los e maior a probabilidade de serem corrigidos corretamente. [1]

A etapa de testes por sua vez visa identificar, nos diversos níveis, o máximo possível de defeitos remanescentes, os quais fariam com que o software entregue não cumprisse com os objetivos desejados.

Testes podem e devem ser aplicados aos diferentes níveis do produto. Para cada nível existe um tipo de teste e cada um envolve quatro etapas: Planejamento, projeto de casos de teste, execução e avaliação dos resultados (Figura 1). Os tipos de testes referentes a cada nível do software são:

- Teste unitário e de integração: É o teste dos menores blocos de um programa, tais como funções, sub-rotinas e procedimentos, além dos erros de interface entre as unidades, quando é feita a integração destes testes.
- Teste funcional: É o processo que busca encontrar discrepâncias entre o programa e sua especificação externa.
- Teste de sistema: O propósito deste tipo de teste é comparar o sistema a seus objetivos originais, logo é fundamental que se tenha um conjunto de objetivos documentados e mensuráveis para o produto. De acordo com Myers [1] há quinze categorias de testes de sistema, dentre os quais podemos citar: Teste de stress, teste de usabilidade, teste de segurança e teste de performance.
- Teste de aceitação: É o processo onde se compara o funcionamento do programa com os seus requisitos iniciais e frente às necessidades atuais dos usuários finais.

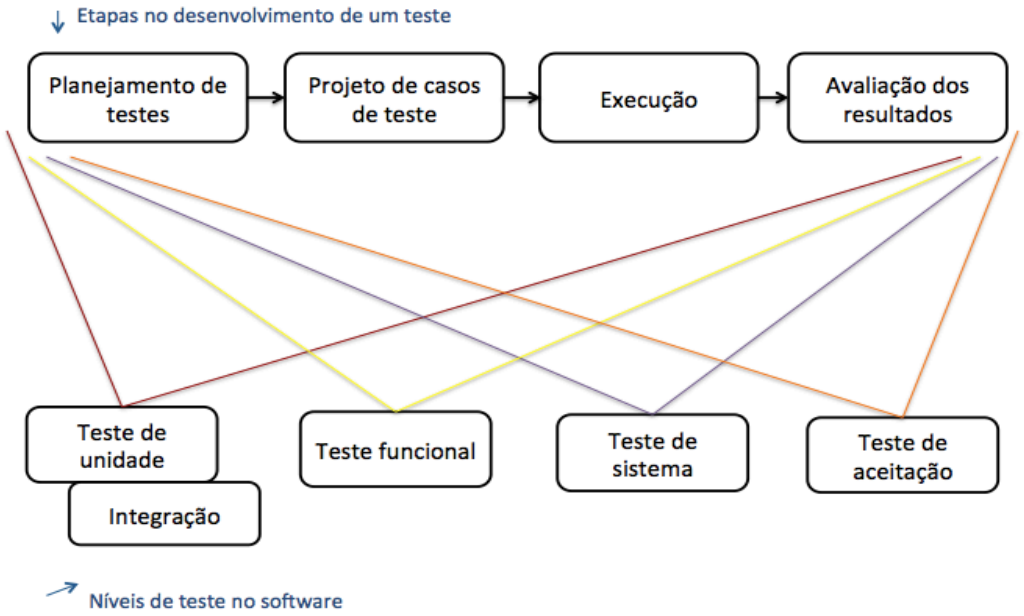

Figura 1

Atualmente a maior parte dos programas apresentam interface gráfica visando facilitar a interação do usuário com o sistema. Aplicações *desktop* ou sistemas *web* são exemplos destes programas. Com o teste na camada de apresentação, também conhecida como *Graphical User Interface* (GUI) deseja-se validar o comportamento da interface gráfica quanto à sua especificação, o que o insere na categoria de testes designada Funcional.

1.2 Testes em GUI

Uma interface gráfica de usuário é uma camada gráfica de apresentação de um software, que aceita como entrada eventos gerados pelo usuário e pelo sistema, produzindo saídas gráficas bem como alterações no estado do código subjacente. Uma GUI contém elementos gráficos, que neste trabalho serão referenciados como *widgets*, e cada elemento possui um conjunto de propriedades. Por exemplo: algumas das propriedades do elemento gráfico (*widget*) ComboBox incluem: Se ele está ativo (active), o número de elementos (numElements) e o elemento selecionado (select). Todos relacionados a um dado instante. Isso implica que em qualquer momento da execução estas propriedades possuem valores, os quais constituem o estado da GUI.

Testar a interface gráfica de um sistema é tão importante quanto testar as outras camadas, pois a GUI é o meio mais direto de interação do usuário com a aplicação, sendo o recurso que mais influencia em como será qualificada posteriormente a experiência de uso pelo usuário. Testar a GUI consiste em verificar os elementos de interface quanto a seus estados, relacionados às propriedades, (p.ex. visível/invisível ou habilitado/desabilitado), valores default, tipo de fonte, layout, cor e resolução, bem como a acurácia do conteúdo numa seção ou elemento. Estas verificações formam o teste funcional da GUI, onde os casos de teste são compostos por sequências de eventos e pelos estados entre os eventos desta sequência.

As abordagens existentes para testes de interface gráfica são [2]:

- A elaboração de casos de testes de maneira *ad hoc* e realizada por um teste manual, onde o testador usa o sistema e verifica visualmente o seu comportamento.
- A implementação manual de scripts de teste, os quais serão executados automaticamente por uma ferramenta adequada.
- O uso de ferramentas *capture/replay*, onde testador executa os fluxos desejados na GUI, um por um, de forma transparente enquanto a ferramenta grava as ações e os estados da interface, gerando os scripts de teste automaticamente. Com uma ferramenta de teste automatizado associada à ferramenta de gravação pode-se posteriormente reexecutar os scripts de forma automática. [3]
- A geração de testes baseada em modelos, ou *Model-Based Testing* (MBT). Nesta modalidade especifica-se o componente de software como algum tipo de modelo formal contendo uma sintaxe definida, a qual viabiliza a geração automática de scripts de teste (formando a suíte de teste). Estes scripts serão a entrada para determinada ferramenta de execução. Além da possibilidade de automação na geração dos casos de teste, o modelo de especificação permite que sejam adotados critérios de teste como uma forma sistemática de avaliar e/ou gerar casos de teste de maior qualidade. [4] Além de especificar como deverá ser testado, o modelo também pode auxiliar o desenvolvedor a produzir um código inicial melhor.

A interação com uma interface gráfica resulta em eventos, que são ações do usuário ou do sistema sobre a interface, e no estado dos elementos gráficos após cada evento. Portanto, eventos e estados definem o comportamento de uma GUI e devem ser considerados em um teste para esta camada. Um modelo de GUI sugere que eventos (também chamados de transições) e estados podem ser valorados e encarados como nós de um grafo que fornece caminhos de execução. Com isso, eventos e estados podem ser reconhecidos e tratados, direcionando o entendimento sobre o comportamento da GUI.

Nesta dissertação são apresentados e avaliados possíveis modelos para a representação de GUIs, visando representar fielmente tanto a sua estrutura quanto o seu comportamento, de forma que se possa gerar casos de teste funcionais rigorosos através de um processo automatizado.

Interfaces gráficas de usuário possuem características peculiares, que serão discutidas adiante, e que requerem modelos específicos para serem descritas com precisão. Logo, uma GUI modelada sem as devidas considerações pode implicar em validações ou suítes de teste incompletas e de baixa qualidade.

Em interfaces gráficas simples, por exemplo um formulário, os eventos de usuário podem obedecer a uma sequência predeterminada e os *widgets* sofrem poucas alterações em suas propriedades, implicando um menor número de estados. Nestas interfaces, modelos pouco sofisticados poderiam resolver o problema de forma satisfatória.

Obter suítes de teste eficazes (boa capacidade em revelar falhas) torna-se um problema quando consideramos GUIs não triviais, onde diversos eventos (transições) estão aptos a ocorrer num determinado instante de forma não-determinística, uma vez que não se sabe a priori onde o usuário irá interagir. Ainda, num sistema de funcionalidades potencialmente concorrentes, a modelagem deve contemplar eventos que poderiam ocorrer simultaneamente. Portanto, é necessário um modelo que permita:

1) A representação dos múltiplos caminhos possíveis a partir de um estado.
2) A representação dos estados de múltiplos objetos. Ou seja, o modelo não deve representar o comportamento apenas de uma única entidade, uma vez que o estado de uma GUI é composto por um conjunto de *widgets* e propriedades associadas.

3) A simulação/execução do modelo, de forma a gerar suítes de teste para a funcionalidade ali representada.

Há atualmente algumas abordagens para a geração de casos de teste para GUI baseados em modelos. O problema nestas abordagens é que os modelos usados não são adequados às características intrínsecas de GUIs mais complexas.

Alguns dos modelos disponíveis são: Máquina de estado, usado em [5]**;** Diagrama de transição de estado; Diagrama de interação e Diagrama de atividades.

Os diagramas de estado [6] e os diagramas de transição de estado têm uma importante limitação. Neles, cada estado é representado por um único item gráfico no modelo e corresponde a um conjunto de atributos que caracterizam um processo ou objeto num dado momento, o que os torna bons na descrição do comportamento de um único processo ou objeto [7]**.** Se considerarmos um sistema (ou uma GUI) de médio porte, podemos dizer ser inviável a construção de um modelo deste tipo, entendendo que numa GUI precisamos verificar o estado de vários elementos de interface. Além disso, em diagramas de estado considera-se que a partir de um estado, apenas uma transição pode estar apta a ocorrer (os eventos são determinísticos). Mesmo que se tenha mais de uma transição saindo do estado, as condições definidas nas transições permitem que apenas uma seja válida. Isso não se adequa a uma GUI, onde a partir de um estado, mais de um evento de usuário ou do sistema pode acontecer.

A Figura 2 abaixo mostra um diagrama de transição de estado. Ele representa os estados de uma garrafa no processo de engarrafamento numa fábrica.

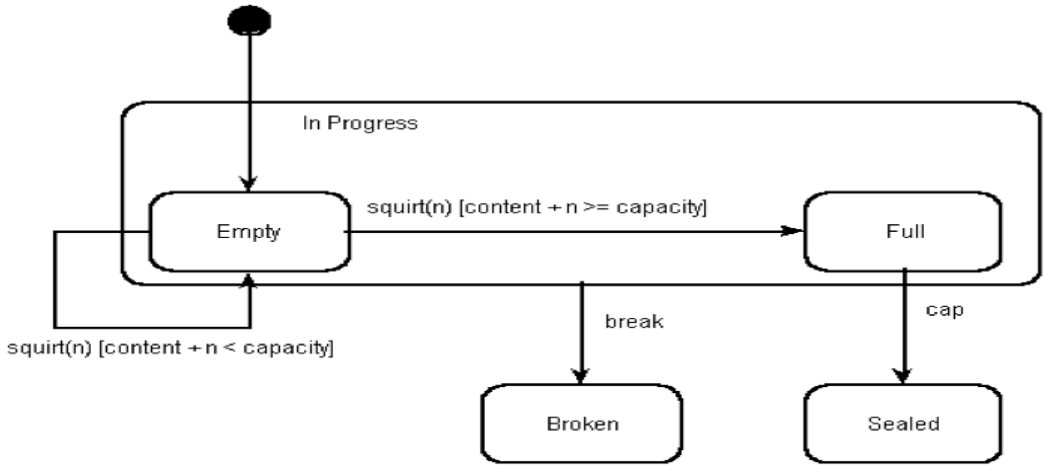

Figura 2 – Exemplo de diagrama de transição de estado. Extraído de [7]

Diagramas de interação descrevem [7] como um grupo de objetos colaboram em algum comportamento, tipicamente um único caso de uso. Uma das suas maiores vantagens é a simplicidade. O ponto fraco é que apesar de descreverem comportamentos, eles não os definem [7]. Não se pode modelar toda a interação e controle necessários para fornecer uma descrição computacionalmente completa. A Figura 3 abaixo é um exemplo deste diagrama.

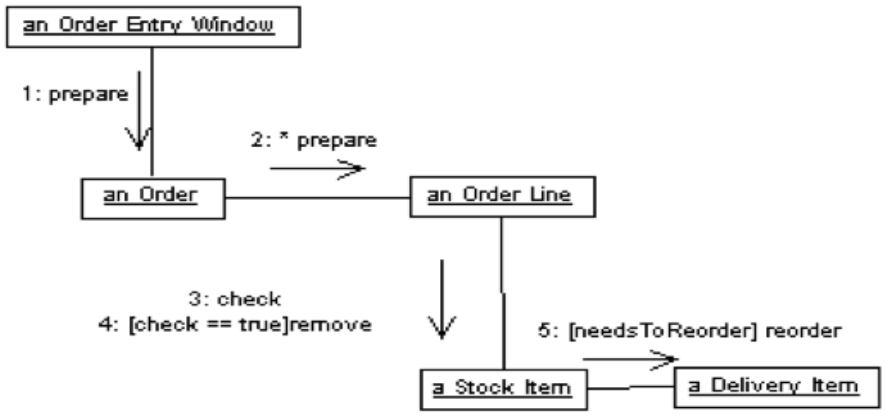

Figura 3 – Exemplo de diagrama de interação. Extraído de [7].

Diagramas de atividades concentram-se em atividades e em suas sequências. É como um fluxograma, mas apoia explicitamente atividades paralelas e sincronizações. São úteis quando se deseja descrever um comportamento paralelo, ou mostrar como comportamentos em vários casos de uso interagem. A desvantagem dos diagramas de atividade é não explicitarem quais objetos executam quais atividades, nem como o serviço de mensagens funciona entre eles. Pode-se rotular cada atividade com o objeto responsável, mas isso não clarifica as interações entre os objetos.

1.3 Motivação

Dadas as soluções existentes menciondas na seção 1.2 para testes de interfaces gráfica de usuário, a abordagem MBT (teste baseado em modelo) foi priorizada no estudo pelos seguintes motivos: Definindo-se um modelo formal, as chances de a suíte ser mais eficaz são maiores, porque temos a estrutura da funcionalidade como um grafo de fluxo de eventos e anotações sobre condições de guarda, permitindo um controle fino sobre a modelagem da GUI quanto aos caminhos disponíveis ao longo do seu comportamento. Ainda, por sua representação estar definida na forma de um grafo, pode-se adotar critérios de cobertura ou

mesmo gerar fluxos aleatórios, que por dispor de todo o modelo da GUI ali representado, encenam a abordagem aleatória de geração de testes. Além disso, com um modelo baseado em redes de Petri pode-se verificar diversas propriedades, tais como *deadlock* e alcançabilidade de um estado.

A geração automatizada de casos de teste a partir do modelo facilita a atualização das suítes de teste à medida que evolvamos requisitos, já que a atualização é realizada diretamente no modelo, e o reuso com o passar do tempo.

Na tentativa de contornar os problemas citados na seção anterior, quanto a modelos adequados à representação de GUIs, identificamos e utilizamos a Rede de Petri (RP), que por suas características especiais demonstrou-se uma boa alternativa. Abaixo são listadas as características mais importantes das Redes de Petri direcionadas ao problema apresentado.

1) A Rede de Petri é capaz de lidar com situações não-determinísticas e de concorrência, permitindo que num determinado estado, mais de uma transição (evento) esteja disponível e sem uma ordem de disparo pré-determinada.
2) O estado da Rede de Petri não é um único elemento gráfico no modelo, mas é determinado pelos *tokens* dispostos pelos lugares da rede. Isso faz com que a RP seja adequada à representação do estado de uma interface gráfica, porque cada lugar na RP pode representar um *widget* quanto a uma propriedade específica e cada *token* entrante traz consigo o valor desta propriedade. Logo, num determinado instante, o estado da GUI é dado por um conjunto de lugares com *tokens*.
3) A Rede de Petri pode ser completamente especificada, com eventos, estados, condições e relações. Também é possível executá-la. Primeiramente de forma manual, validando visualmente o comportamento da funcionalidade modelada e a especificação. Depois a execução automática permite a geração da suíte de teste.

A geração dos casos de teste a partir de uma Rede de Petri se dá através da simulação. Nela, cada caminho executado na rede é uma sequência de transições e estados, representando ações e oráculos, respectivamente. Dessa forma, cada sequência executada é um caso de teste.

A simulação pode ser executada passo a passo pelo testador, onde a cada passo ele escolhe qual transição, entre as que estão habilitadas, irá disparar. Ou a

execução pode ser automática e aleatória, que é a abordagem mais interessante no ponto de vista deste trabalho e será discutida com mais detalhes no Capítulo 4.

A execução aleatória da rede é baseada numa técnica de teste que consiste na geração de dados de entrada aleatórios. No caso de interface gráfica, as entradas são os eventos disponíveis concorrentemente a cada instante, assim como os parâmetros nas transições. Este tipo de teste é capaz de fornecer naturalmente, sequências de eventos incoerentes com as executadas pelo usuário acostumado com o software ou mesmo com o testador que tende a escrever testes que seguem pelo chamado *happy path,* caminho semelhante ao uso comum e esperado. Por isso, testes aleatórios, com seus casos de teste longos e mesmo repetitivos, sugerem boas chances em detectar problemas.

Para avaliar a eficácia e a eficiência obtidas com a abordagem proposta, aplicamos o método ao software V3O2/Tecgraf, comparando o tempo e a viabilidade deste com a escrita de scripts de teste manualmente, avaliando o número de defeitos encontrados e o escore de mutação da Análise de Mutantes (AM), o qual representa uma medição para a eficácia.

O software V3O2 foi usado como prova de conceito para o estudo deste trabalho. O V3O2 dá suporte a geólogos e geofísicos ao longo de todo o ciclo de vida dos dados sísmicos na cadeia produtiva da empresa [8]. Naturalmente, ele é baseado em interface gráfica para apoiar as tarefas tanto de visualização quanto de operações sobre diversos tipos de dados sísmicos. Uma característica das interfaces no V3O2 é que são bastante variadas e dinâmicas, permitindo uma série de interações não ordenadas e sujeitas também a eventos concorrentes. Por exemplo, pode-se estar operando em uma janela específica de uma funcionalidade enquanto ocorre o carregamento de um dado. Quando este carregamento terminar, ocorrerá um evento de término, que irá "avisar" a janela, atualizando algum(ns) de seu(s) componente(s). No capítulo 5, o software de estudo é mais detalhado focando também nas funcionalidades testadas neste trabalho.

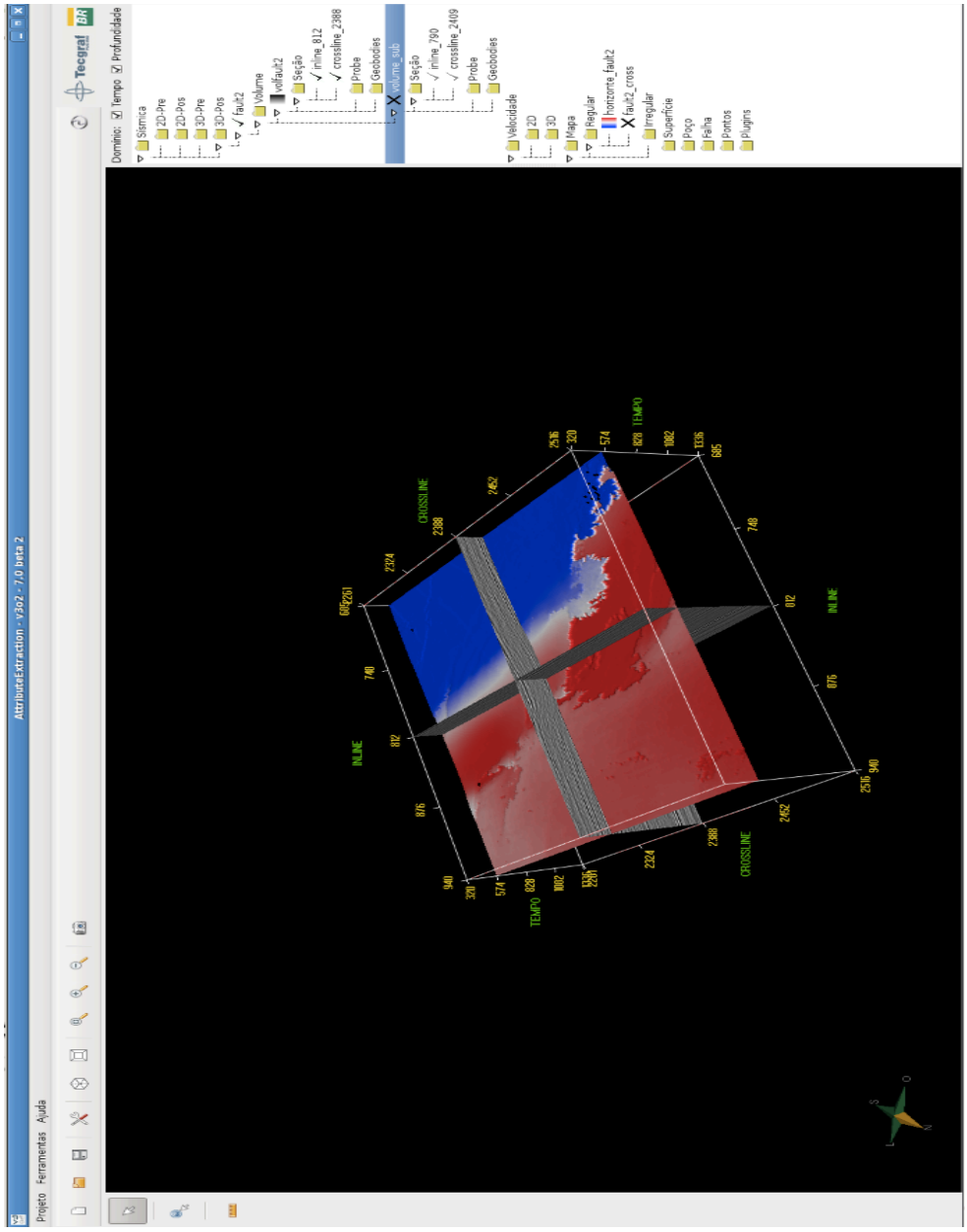

Figura 4a – Interface principal do V3O2

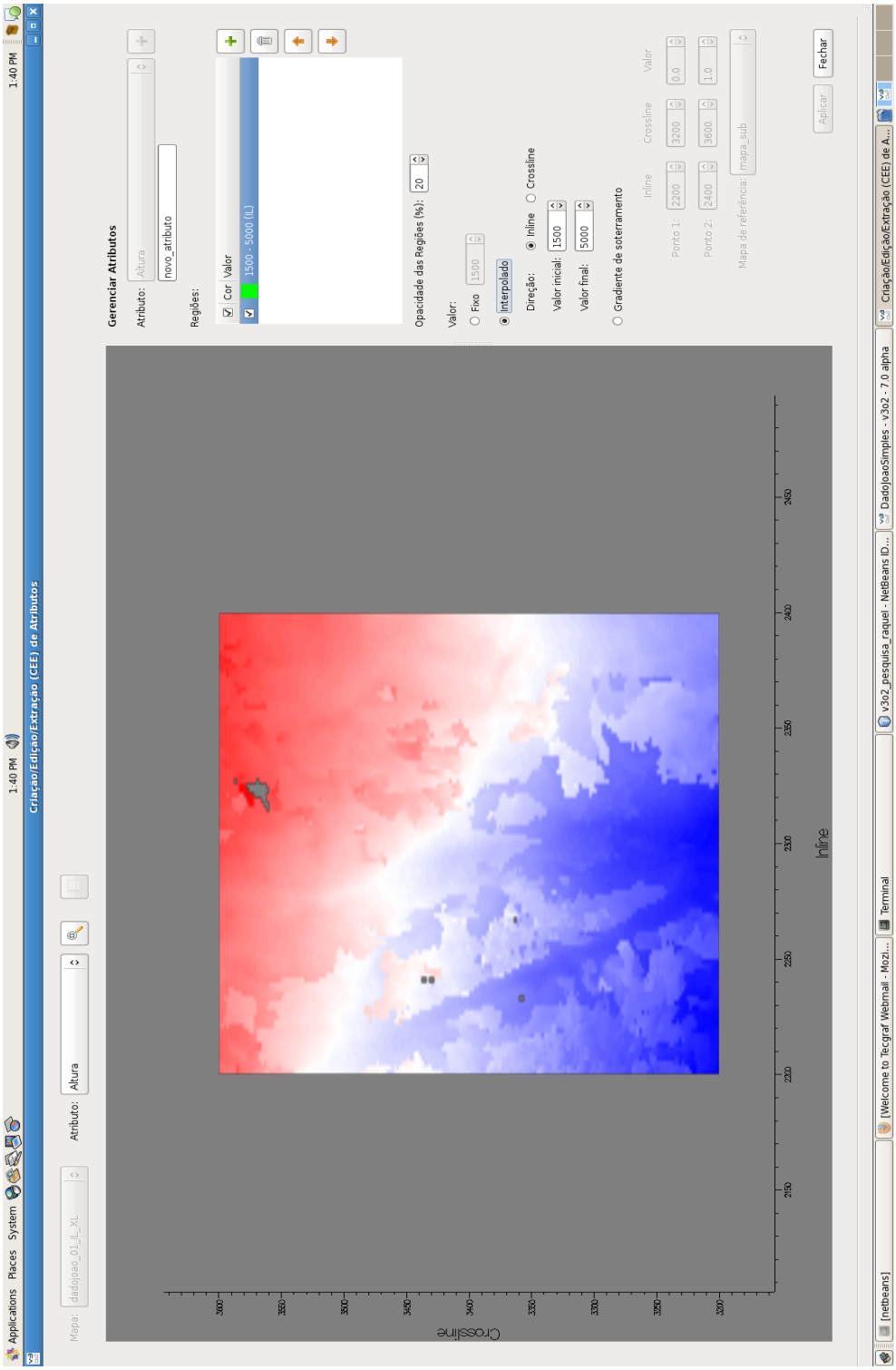

Figura 4b – GUI de uma sub funcionalidade

Para complementar o processo de teste deste estudo foi necessário desenvolver uma ferramenta, denominada guiftG – Graphical User Interface Functional Test Generator, para interpretar a Rede de Petri modelada e as simulações executadas, disponíveis em arquivos xml e texto, e transformá-los em suítes de teste no

formato de scripts executáveis. A execução dos testes é realizada por uma ferramenta de teste funcional já existente no software de estudo. Dessa forma, após a construção e a validação do modelo, temos geração e execução automáticas de scripts de teste.

Neste trabalho também foi desenvolvida uma ferramenta que aplica o critério de teste Análise de Mutantes no SUT (*Software Under Test*) para medir a eficácia da suíte de testes gerada. No capítulo 5 a Análise de Mutantes será introduzida e detalhada. A eficácia é dada pelo número de mutantes mortos dividido pelo número total de mutantes não equivalentes gerados. Esse é o escore de mutação, que mede a eficácia e ajuda a incrementar os casos de teste na suíte. Como os mutantes são gerados com operadores que simulam erros no código do SUT, os que permanecem vivos indicam a falta de casos de teste capazes de observar aquele tipo de erro.

A Figura 5 abaixo, é o esquema do processo deste trabalho.

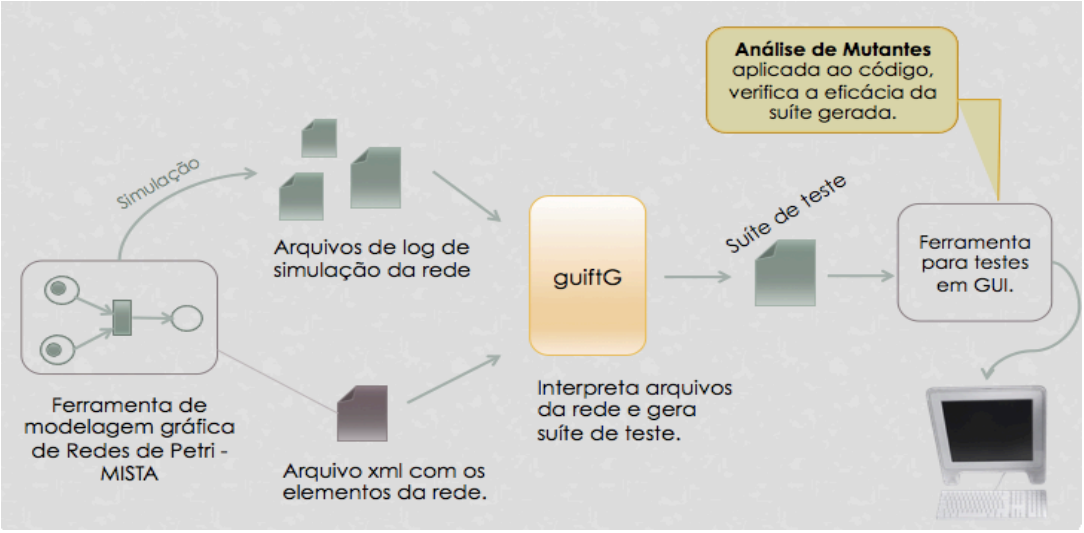

Figura 5 – Fluxo do processo aplicado neste trabalho

Quatro funcionalidades do V3O2 foram utilizadas como experimento. Sendo assim, foram modeladas como Rede de Petri e simuladas. As suítes geradas, foram executadas na ferramenta de teste funcional e verificadas quanto a sua eficácia com a Análise de Mutantes.

De uma maneira geral, as maiores vantagens identificadas nesse processo foram:

- O potencial da Rede de Petri para modelar interfaces gráficas: Eventos são não determinísticos e estados são distribuídos pela rede.
- Capacidade de simulação, verificando o comportamento e gerando casos de teste de maneira eficiente.

As desvantagens observadas foram:

- Dependendo do tamanho da funcionalidade a rede pode se tornar bastante grande. A semântica adotada inicialmente foi tal que os lugares da RP representam um *widget* e sua propriedade (e um *token* nesse lugar contém o valor dela). Com isso, a ferramenta guiftG foi toda desenvolvida com esse propósito. Porém, uma outra possibilidade de modelagem é um lugar representar apenas o *widget* e utilizarmos *tokens* estruturados, que ao invés de conterem um único valor, seriam a tupla <propriedade,valor>. Esse formato permitiria uma redução no tamanho da rede.
- Como a suíte executável é gerada automaticamente a partir de um modelo com limitações de sintaxe e representação, acaba sendo necessário realizar ajustes ou incrementos nos scripts gerados. Essa limitação é bastante comum num processo automático e para que seja amenizada é preciso que a ferramenta interpretadora seja bem aprimorada.

De uma maneira geral, os resultados obtidos sugerem que o método apresentado neste trabalho pode ser útil como processo de teste a ser adotado em um ambiente de desenvolvimento. Tanto pelas características intrínsecas da RP de Alto Nível e pela forma de representação de GUIs sugerida nesse trabalho, quanto pelos resultados sobre a sua eficiência. Os resultados obtidos para a eficácia do método refletem a sua capacidade em detectar defeitos, porém os valores obtidos para o escore de mutação através da Análise de Mutantes não foram altos: Nas simulações automáticas, ficaram entre 14,4% e 36,7%. Apesar disso, os baixos escores não são suficientes para invalidar o método, pois as razões identificadas são contornáveis, estando relacionadas à evolução das ferramentas. Essas razões são discutidas em mais detalhes no Capítulo 6.

Os requisitos considerados neste trabalho para que os resultados atendessem às expectativas mais elementares sobre testes em GUI são:

- A criação das suítes de teste executáveis não ser extremamente custosa.

- A execução das suítes de teste não ser extremamente custosa.
- Conseguir suítes de teste que sejam eficazes em revelar problemas.

Isso se resume a conseguir o menor número casos de teste com maior potencial de desvendar problemas.

Apesar de a elaboração do modelo formalizado demandar um certo tempo, o custo de formulação do modelo é absorvido pela rapidez da geração e pelo rigor da suíte gerada.

Não foi considerado como requisito, que o processo e as ferramentas para a construção de casos de teste sejam de fácil aprendizagem, já que o custo para isso é contabilizado apenas uma vez, por grupo de pessoas escaladas para desenvolvimento e testes. E mesmo com a entrada eventual de novos desenvolvedores, entende-se que o custo é compensado com o uso constante de um bom processo de teste.

Visamos então obter uma melhora na descoberta de problemas e na praticidade e eficiência de testes em funcionalidades de GUI, frente ao método utilizado atualmente no software real usado como prova de conceito deste trabalho.

1.4 Estrutura da Dissertação

Esta dissertação está organizada da seguinte forma: No capítulo 2 apresentamos a Rede de Petri, seus tipos e suas características. Introduzimos a ideia da representação de uma GUI na Rede de Petri, apresentando um exemplo já com a semântica e a sintaxe definidas nesse trabalho. No capítulo 3 consta a revisão bibliográfica, com os trabalhos relacionados ao assunto testes baseados em modelo, abordando outros tipos de modelos e Redes de Petri. São expostos também trabalhos relacionados a ferramentas para testes em GUI existentes. Por fim, falamos de trabalhos relacionados à técnica Análise de Mutantes, dando ênfase na aplicação desta técnica ao nível funcional e não somente ao unitário, como é tradicional. No capítulo 4 falamos sobre o método proposto neste trabalho, detalhando o uso das ferramentas desenvolvidas e a execução das suítes geradas pelo modelo da RP, no SUT V3O2. Uma visão geral sobre o software de estudo V3o2 também é dada neste capítulo. Dá-se ênfase ainda à técnica de modelagem da GUI como uma Rede de Petri na ferramenta de modelagem e aos detalhes a que deve-se ter em mente. No capítulo 5 explica-se o critério de teste Análise de Mutantes e o ob-

jetivo da sua adoção na metodologia deste trabalho. No capítulo 6 descreve-se a aplicação do processo de teste proposto em quatro funcionalidades selecionadas do SUT e os resultados são apresentados. Por fim, é realizada a análise sobre os resultados e descrevem-se as considerações finais sobre a aplicação prática. O capítulo 7 encerra esta dissertação, apresentando as conclusões, as limitações deste trabalho e os trabalhos futuros de pesquisa acerca deste tema ou de desenvolvimento na forma de incrementos sobre o método proposto.