5 Critério de teste Análise de Mutantes

A técnica de teste Baseada em Erros, ou mutantes, utiliza informações sobre os tipos de erros mais frequentes no processo de desenvolvimento de software para conduzir a geração de testes. Critérios típicos desta técnica são Semeadura de Erros e Análise de Mutantes [30]. Este último surgiu na década de 70 na Yale University e Georgia Institute of Technology e segundo Maldonado et al. em [30] um dos primeiros artigos que descrevem a ideia foi publicado em 1978.

A razão que nos levou a aplicar a análise de mutantes sobre as suítes de teste geradas por Rede de Petri nesse trabalho, é podermos avaliar a eficácia destas suítes. A eficácia é o percentual dos defeitos existentes que a suíte é capaz de identificar. O problema nessa medição é que não sabemos quais são os erros existentes no programa, mas com o uso de mutantes pode-se ter uma boa estimativa.

A análise de mutantes é fundamentada em duas hipóteses: A do programador competente e a do efeito de acoplamento.

A primeira assume que programadores experientes escrevem programas próximos do correto, de forma que os defeitos são facilmente identificados por casos de teste adequados. Assumindo a validade desta hipótese, pode-se afirmar que os erros introduzidos em um programa são pequenos desvios sintáticos, que o leva a um comportamento incorreto.

A segunda hipótese assume que erros complexos estão relacionados a erros simples. Um corolário para ela é que, se cada defeito simples for considerado e descartado, pode-se assumir que os defeitos complexos também o serão. Alguns estudos empíricos confirmaram essa hipótese. [30]

A análise de mutantes é aplicada da seguintes forma: Primeiramente um programa P é testado com um conjunto de casos de teste T criado para ele. À medida que falhas vão sendo descobertas pelo teste, os defeitos causadores são corrigidos, até quando o conjunto de teste T não revelar mais problemas. Nesse ponto, o programa P ainda pode conter defeitos que o conjunto T não conseguiu revelar. P sofre então pequenas alterações dando origem a programas P', "próximos" de P, mas com defeitos simples consistindo de pequenas alterações sintáticas. P1, P2, ..., Pn são mutantes de P. Para introduzir os defeitos, gerando cada um desses mutantes, operadores de mutação devem ser definidos com base nos defeitos típicos inje-

tados por programadores. A seguir, cada programa mutante P' é executado por vez, com o conjunto de testes T. Se T for capaz de revelar o defeito introduzido em P', falhando o teste, diz-se que o mutante foi morto. O objetivo é que todos os mutantes sejam mortos por T, porque quando algum mutante permanece vivo, significa que o conjunto de teste T é incapaz de revelar o erro causado pelo defeito no ponto onde houve a mutação. Com isso, T deve ser incrementado de forma que consiga matar aquele mutante. Entretanto, alguns mutantes podem não ser observáveis, pois para qualquer execução possível o resultado com mutante ou sem mutante será o mesmo. Tais mutantes são conhecidos por mutantes equivalentes e contribuem nada do ponto de vista avaliar a eficácia do teste. Isso significa que todos os mutantes que não foram mortos precisariam ser analisados a fim de determinar se são equivalentes ou não. Os mutante não equivalentes são mutantes úteis. Como determinar a equivalência entre dois programas é indecidível, essa tarefa precisa ser realizada à mão possivelmente com apoio de heurísticas. Assim, determinar a equivalência de um mutante tende a ser muito caro [24], por isso neste trabalho o parâmetro EM(P) foi estimado em 10% dos mutantes gerados, tendo como base o trabalho empírico realizado em [22].

Essa estimativa, entretanto, foi otimista quando considerados outros trabalhos relacionados, que segundo Jia e Harman em [32] relatam mutantes equivalente com percentuais entre 10% e 40%. Um percentual baixo de mutantes equivalentes equivocadamente adotado, contribui para um escore de mutação baixo, como pode ser conferido na fórmula mais abaixo. Esta questão será mais discutida nos capítulos 6 e 7.

Partindo destas duas hipóteses, pode-se assumir que: Se são gerados mutantes para todos os defeitos simples possíveis cometidos por programadores, temos o conjunto dos defeitos possíveis do programa. Logo, após a execução da análise de mutantes e a obtenção de quantos foram os mutantes mortos pelo conjunto de testes, podemos calcular o percentual dos defeitos existentes que a suíte é capaz de encontrar.

Em [31], Demillo fornece uma medida objetiva para a adequação dos casos de teste do programa P, definindo um escore de mutação (*mutation score*), que é a proporção dos mutantes mortos dentre os de mutantes úteis gerados (excluindo-se os equivalentes). Portanto, o escore de mutação varia no intervalo entre 0 e 1 e

quanto maior o escore, maior a eficácia da suíte de testes gerada. A fórmula é dada abaixo:

$$\boldsymbol{ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)}}$$

onde:

P → Produto

T → Conjunto de casos de teste inicial

DM(P, T) → Número de mutantes mortos por T

M(P) → Número total mutantes gerados

EM(P) → Número de mutantes equivalentes

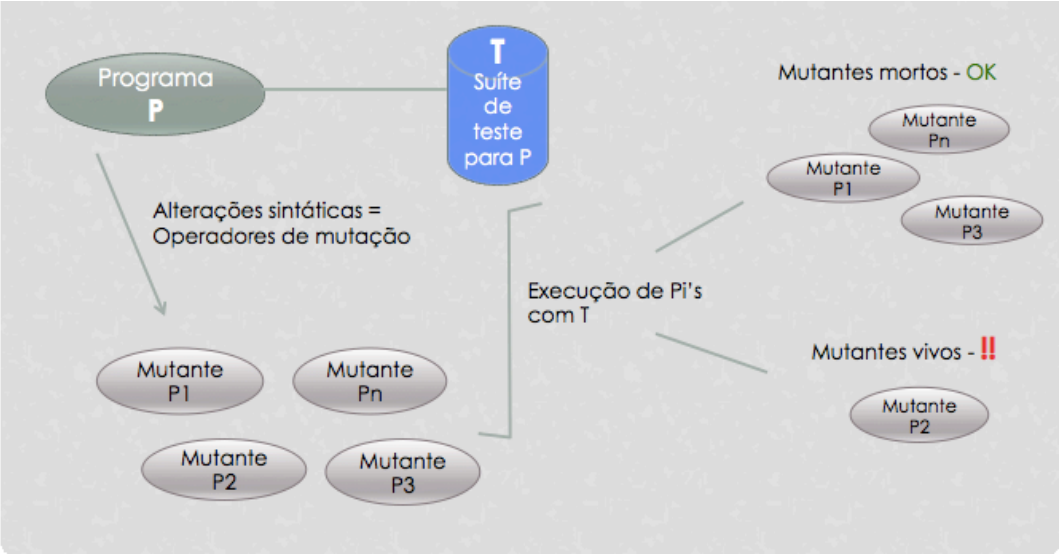

Figura 27 – Esquema da Análise de Mutantes em um programa P

5.1 AM tradicional

Originalmente a análise de mutantes é aplicada ao nível de unidade (funções e métodos) e os operadores criados para esse nível representam erros típicos cometidos por programadores numa unidade de software. Estes operadores de mutação alteram: Variáveis, operadores, constantes e atribuições. [32].

Operadores de mutação voltados para o nível de integração também foram estudados e apresentados pela primeira vez em [33]. No nível de integração, os operadores são definidos de forma a representar erros cometidos na conexão entre unidades do programa.

Já a mutação no nível de classe foi alvo de pesquisas que tentavam criar operadores voltados para características típicas de programas orientados a objetos,

tais como polimorfismo e herança. No entanto, em [22] e [23] revelou-se que operadores para o nível de classe geraram uma porcentagem alta de mutantes equivalentes, (45% e 86% respectivamente, quando a média para operadores tradicionais é de 13%) indicando que o uso destes operadores pode ser uma estratégia ruim.

Existem ainda operadores de mutação que foram definidos visando defeitos ou vulnerabilidades específicas de programas em C, como erros associados com alocações estática e dinâmica. [32].

5.2 AM no nível funcional e de sistema

Em [24] foi introduzido o conceito de mutação no nível funcional e de sistema, onde o objetivo é a detecção de defeitos originados por uma má construção da lógica das operações como um todo. Para isso, definem operadores para o nível do sistema, dentre os quais pode-se citar: Operadores relacionados a configuração, interação entre classes e GUIs.

Como neste trabalho os casos de teste são gerados a partir de Rede de Petri representando uma interface gráfica, temos que a natureza dos testes é de nível funcional, abrangendo interações entre classes e código específico da lógica de interface. Adotamos então, na análise de mutantes, além das mutações de nível de unidade, as mutações de nível funcional. Os operadores do nível da unidade para C++ foram coletados da literatura existente. Os operadores do nível funcional foram definidos de acordo com as características de implementação do software sob teste e serão apresentados na seção 5.3.1 deste capítulo.

Como visto, diversos tipos de operadores podem ser definidos para a análise de mutantes, cada qual voltado para um nível de teste. Para um conjunto de testes unitários, as mutações devem ser de nível unitário. Para um conjunto de testes funcionais, o ideal é que as mutações simulem problemas de interação entre objetos e problemas na lógica da GUI. Mas além disso, os tradicionais operadores do nível de unidade da linguagem em questão também podem ser usados, porque o código continua sujeito a este tipo de erro.

5.3 Análise de Mutantes no SUT

Para aplicar a análise de mutantes sobre o código do SUT e verificar a eficácia da suíte de teste gerada foi preciso implementar uma ferramenta que trabalhasse não só com os operadores tradicionais da linguagem C++, mas também com

os operadores de nível funcional característicos do programa, de forma que os mutantes gerados simulassem defeitos de lógica específicos e tratamentos padrões no sistema ao qual foi aplicado.

5.3.1 Operadores

Uma pesquisa foi realizada a fim de obter um conjunto extenso de operadores de mutação para C++. Inicialmente, aproximadamente trinta foram selecionados, abrangendo os níveis de unidade, integração, lógica de interface gráfica e sistema. No entanto, devido à alta complexidade em implementar a mutação do código para alguns tipos de operadores, foi preciso excluir alguns deles da seleção inicial, tornando essa ferramenta mais próxima de um protótipo.

A quantidade final de operadores de mutação usados foi catorze. A seguir eles estão listados e organizados pelo seu tipo:

```
/** C++ nível unitário */
ROR,      //Substitui operadores relacionais e de igualdade (< , > , <= , >= , == , !=)
LOR,      //Substitui operadores lógicos && e || um pelo outro, um de cada vez
AOR,      //Substitui operadores aritiméticos. (+, - , * , / , += , -= , *= , /=) /* +,-
BVR,      //Substitui valores boleanos true e false um pelo outro.

/** C++ nível de integração */

/** C++ nível de sistema */
LRQD,    //Remove chamadas lockRequest(). Uma por vez
LRLD,    //Remove chamdas lockRelease(). Uma por vez
AOD,     //Remove chamadas adicionaObservador(). Uma por vez
ROD,     //Remove chamadas removeObservador(). Uma por vez

/** GUI/Gtk */
GTK_SRSD,  //Remove as chamadas a setRangeSelection
GTK_SSND,  //Remove as chamadas a setSensitive
GTK_SSLD,  //Remove as chamadas a setSelected
GTK_SID,   //Remove as chamadas a selectItem
GTK_SSNC,  //Substitui os argumentos em setSensitive(bool)
GTK_SSLC,  //Substitui os argumentos em setSelected(bool)
```

Figura 28 – Operadores de mutação utilizados na ferramenta-protótipo desenvolvida

Durante a execução da ferramenta de análise, cada um desses operadores pode gerar vários programas mutantes, já que pode haver diversas ocorrências no código, da alteração especificada pelo operador. Como já mencionado, um mutante é um programa semelhante ao original, mas com uma alteração que produz um erro.

5.3.2 Ferramenta desenvolvida

A ferramenta desenvolvida foi denominada MATool e seu modelo e funcionamento estão descritos abaixo.

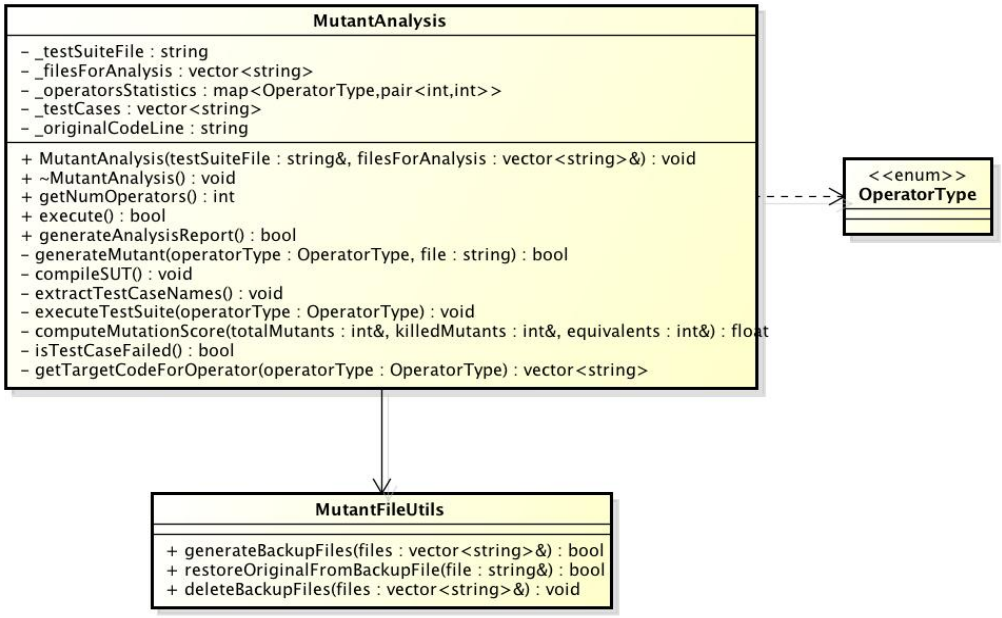

Figura 29 – Diagrama de classes da ferramenta MATool

Como pode ser visto na Figura 29, a ferramenta é composta pela classe principal MutantAnalysis, que é responsável por todo ao algoritmo da análise de mutantes (descrito abaixo), desde a geração dos mutantes até a geração de um relatório final. Esta classe contém o enum OperatorType, que é onde estão listados os operadores de mutação da figura 28. A classe MutantAnalysis tem como suporte o namespace MutantFileUtils, que auxilia nas tarefas de gerar e restaurar cópias dos arquivos de classe que são constantemente alterados ao longo da execução.

A MATool é executada por linha de comando, com o usuário fornecendo um arquivo com a suíte de teste e um ou mais arquivos fontes das classes a serem mutadas. Em linhas gerais o algoritmo da ferramenta é o seguinte: Para cada arquivo de classe a ser analisada e para cada um dos 14 operadores de mutação, realiza-se uma busca por pontos onde pode ocorrer a mutação e, enquanto houver mutações para aquele operador, cria-se um mutante. Com este mutante, compila-se o código novamente, e executam-se os testes da suíte com aquele novo executável mutante. Se houver falha no teste, o mutante foi morto e isso é computado.

Ao final da execução da ferramenta, um relatório é gerado com o resultado na análise. Um exemplo desse relatório, relativo a uma funcionalidade do SUT, encontra-se na Figura 30 abaixo.

```
Relatório da Análise de Mutantes

Classes sob mutação:
../v3o2_pesquisa_raquel/v3o2/src/horizon/cutting/CutHorizonPresenter.cpp
../v3o2_pesquisa_raquel/v3o2/src/horizon/cutting/CutHorizonArea.cpp

Suíte de teste:
../v3o2_pesquisa_raquel/v3o2/src/horizon/CutHorizonUITest.cpp


Número de operadores de mutação: 14

Estatísticas de operadores e mutantes:
OPERADOR         MUTANTES GERADOS        MUTANTES MORTOS
ROR                      4                       1
LOR                      2                       0
AOR                      5                       0
BVR                      16                      6
LRQD                     0                       0
LRLD                     0                       0
AOD                      0                       0
ROD                      0                       0
GTK_SRSD                 0                       0
GTK_SSND                 3                       2
GTK_SSLD                 2                       1
GTK_SID                  0                       0
GTK_SSNC                 0                       0
GTK_SSLC                 1                       1


Escore de Mutação: (Mut. mortos / Mut. gerados - Equivalentes)
Total de mutantes gerados: 33
Total de mutantes mortos: 11
Mutantes equivalentes: 3


Escore de Mutação = 36.6667%
```

Figura 30 – Relatório gerado pela MATool, indicando a avaliação de eficácia de uma suíte executada no SUT.

5.3.3 Execução da AM no SUT

Junto a outros critérios de avaliação, como eficiência, da metodologia proposta nesse trabalho, a análise de mutantes teve como objetivo avaliar a eficácia das suítes de teste geradas a partir de modelos de funcionalidades do software em teste, através do escore de mutação de cada suíte.

Quatro funcionalidades do SUT V3O2 foram usadas como experimento neste trabalho e para cada uma temos uma suíte de teste gerada. A análise de mutantes foi aplicada para cada suíte e o tempo de execução foi de aproximadamente uma hora em cada uma. Se a ferramenta de análise for evoluída futuramente, sendo

mais completa em termos de operadores, o tempo de execução crescerá consideravelmente. Contudo, esse tempo pode ser desprezado, porque a análise acontece apenas uma vez. Depois de analisada pela AM, a suíte irá se juntar aos outros testes na integração contínua, mas a verificação da sua eficácia acontece, a princípio, apenas uma vez.

No próximo capítulo, Experimentação do Processo Proposto, as estatísticas e os resultados obtidos na aplicação da metodologia deste trabalho serão apresentados, inclusive os resultados da análise de mutantes.