UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

CARLOS EDUARDO SIQUEIRA

ANÁLISE DE COBERTURAS DE TESTE PARA EXTRAÇÃO DE MODELOS DE COMPORTAMENTO

Trabalho de Graduação de Curso apresentado como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação da Universidade Federal do Rio Grande do Sul

Orientação: Prof. Dr. Lucio Mauro Duarte

Porto Alegre, dezembro de 2011.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do CIC: Prof. Raul Weber
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço à Universidade Federal do Rio Grande do Sul e ao Instituto de Informática pelo ensino e a oportunidade.

Agradeço ao Professor Doutor Lúcio Mauro Duarte pela orientação e auxílio durante todas as etapas deste trabalho.

Agradeço à minha Família e Amigos pelo apoio e incentivo.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 5
LISTA DE FIGURAS 6
LISTA DE TABELAS 7
RESUMO 8
ABSTRACT 9
1. INTRODUÇÃO 10
1.1. Abordagem do Tema 10
1.2. Motivação 10
1.3. Objetivos 11
1.4. Organização do Texto 11
2. REFERENCIAL TEÓRICO 12
2.1. Critérios de Cobertura de Teste 12
2.1.1. Line Coverage 12
2.1.2. Branch Coverage 12
2.1.3. Condition Coverage 13
2.1.4. Path Coverage 13
2.1.5. Random Coverage 14
2.2. Modelos de Comportamento 14
2.3. Análise de Cobertura 15
2.3.1. EclEmma 15
2.3.2. Eclipse CFG Generator 15
2.4. Extração de Modelos 16
2.4.1. LTSE 17
2.4.2. LTSA 17
3. EXTRAÇÃO DE MODELOS DE COMPORTAMENTO 18
3.1. Casos de Teste 18
3.2. Instrumentação 20
3.3. Geração de Logs 22
3.4. Extração do Modelo de Comportamento 23
3.5. Outros Experimentos 24
4. ANÁLISE DE COBERTURA DE TESTE X MODELO DE COMPORTAMENTO 29
4.1. Critérios 29
4.1.1. Critério Estrutural 29
4.1.2. Critério Observacional 30
4.2. Discussão de Resultados 31
4.2.1. Coberturas de Teste 32
4.2.2. Relação Custo-Benefício 32
4.2.3. Análise dos Sistemas Estudados 32
4.3. Trabalhos Correlatos 33
5. CONCLUSÃO 34
6. REFERÊNCIAS 35
APÊNDICE 36

LISTA DE ABREVIATURAS E SIGLAS

CFG	Control Flow Graph
FSP	Finite State Process
LTSA	Labelled Transition System Analyzer
LTSE	Labelled Transition System Extractor
TXL	Turing eXtender Language

LISTA DE FIGURAS

Figura 2.1: Exemplo de CFG gerado pelo Eclipse CFG Generator 16
Figura 2.2: Exemplo de modelo FSP (esquerda) e o LTS gerado (direita) 17
Figura 3.1: Diagrama de Etapas da Extração de Modelos 18
Figura 3.2: Código de *TrafficLights* 19
Figura 3.3: CFG de *TrafficLights* 20
Figura 3.4: Modelo de *TrafficLights* para a cobertura *Condition Coverage* 23
Figura 3.5: Código de *ACController* 25
Figura 3.6: Casos de teste e modelo de *ACController* para *Line Coverage* 26
Figura 3.7: Casos de teste e modelo de *ACController* para *Branch Coverage* 26
Figura 3.8: Casos de teste e modelo de *ACController* para *Condition Coverage* 27
Figura 3.9: Casos de teste e modelo de *ACController* para *Random Coverage* 27
Figura 3.10: Casos de teste e modelo de *ACController* para *3-Path Coverage* 28

LISTA DE TABELAS

Tabela 3.1: Suíte de testes para *Condition Coverage* em *TrafficLights* 19

Tabela 4.1: Resultados para *TrafficLights* 29

Tabela 4.2: Resultados para *Editor* 30

Tabela 4.3: Resultados para *ACController* 30

Tabela 4.4: Resultados para *Bounded Buffer* 30

RESUMO

O presente trabalho foi realizado considerando-se a importância de Testes de Software e a Extração de Modelos de Comportamento em áreas como a Engenharia de Software. Especificamente, focou-se na extração de modelos a partir de código pré-existente, buscando alcançar o modelo de comportamento mais completo para os códigos tratados. A metodologia adotada abrange a geração casos de testes para atender cinco diferentes critérios de cobertura de teste, a análise estática e dinâmica de códigos pré-existentes, a instrumentação desses códigos, a extração dos modelos de comportamento e a análise estrutural e observacional dos modelos gerados. Como resultado final, foi identificado um critério de cobertura de testes que, apesar de possuir um conjunto de casos de testes maior que os demais, gera os modelos mais completos em relação aos demais para os sistemas estudados.

Palavras-Chave: Verificação de Modelos de Comportamento, Extração de Modelos de Comportamento, Critérios de Cobertura de Teste.

ABSTRACT

The present work was developed considering the importance of Software Testing and Behaviour Model Extraction in areas such as Software Engineering. Specifically, it was focused on model extraction from existing code seeking to achieve the most complete model for some sample codes. The methodology includes generating test cases to attend five different testing coverage criteria, static and dynamic analysis of pre-existing code, the instrumentation of these codes, the extraction of behaviour models and observational and structural analysis of the generated models. As a final result, it was identified a criterion for testing coverage that, despite having a set of test cases larger than the others, generates most complete models for the discussed systems.

Keywords: Model Checking, Behaviour Model Extraction, Testing Coverage Criteria.

1. INTRODUÇÃO

1.1. Abordagem do Tema

Garantir correção de sistemas de software é muito importante, visto que eles fazem parte do cotidiano de todos. Nesse sentido, Teste de Software (PEZZÈ, 2008) e Verificação de Modelos (CLARKE, 1999) são técnicas bem difundidas para o aumento da confiança na correção de sistemas. Em relação à Verificação de Modelos, existe a possibilidade da avaliação exaustiva de um modelo em busca de inconsistências em relação a uma especificação. Além disso, a existência de um modelo permite diversos tipos de outras análises e também uma melhor compreensão sobre como o sistema funciona. No entanto, a extração de modelos de comportamento não é um processo trivial pelo fato dele ser construído de forma manual, ou seja, está propensa a inserção de erros por parte do programador.

Como uma forma de prover esse modelo, técnicas de Extração de Modelos (HOLZMANN, 1999) foram propostas, tais como (COOK, 1998) e (LORENZOLI, 2006). O processo da extração de modelos é baseado na geração de um modelo a partir de um código pré-existente. A abordagem proposta em (DUARTE, 2007) utiliza uma combinação de informação estática com informação dinâmica para a geração do modelo a partir de rastros de execução gerados pelo sistema. Tal abordagem, já demonstrou gerar modelos *corretos* em relação à implementação correspondente (DUARTE, 2008), i.e., o modelo não inclui comportamentos inválidos porque eles são gerados a partir de rastros de informações obtidos através de execuções reais do sistema. No entanto, tal abordagem não garante a *completude* do modelo em relação à implementação dos sistemas modelados, o quê significa que certos comportamentos válidos podem não estar presentes no modelo.

1.2. Motivação

Dada a restrição da abordagem apresentada em (DUARTE, 2007), em relação à completude dos modelos gerados, pensou-se em utilizar um conjunto de testes como forma de selecionar quais rastros de execução comporiam o modelo, visando a aumentar a completude do sistema. Para isto, deveria ser definido como selecionar os casos de teste de forma a cobrir a maior parte possível do comportamento do sistema. Nesta direção, esse trabalho busca analisar a influência de critérios para seleção dos testes para geração dos rastros sobre os modelos gerados, considerando-se critérios relativos à completude do sistema.

1.3. Objetivos

O objetivo principal deste trabalho é realizar o exercício de cinco diferentes critérios de cobertura de teste (PEZZÈ, 2008) - *Line Coverage*, *Branch Coverage*, *Condition Coverage*, *Path Coverage* e *Random Coverage* - sobre quatro sistemas simples retirados de (DUARTE, 2007) que, em sua maioria, fornecem uma interface para o recebimento interativo de entradas do usuário. Além disso, visa-se determinar um tipo de cobertura de teste que gere o modelo de comportamento mais completo possível entre os critérios utilizados. Tal análise deve basear-se em critérios que definam como comparar a completude de diferentes modelos de comportamento.

A análise quanto à completude do modelo de comportamento é realizada de duas maneiras: através da análise do número de estados e transições, onde quanto maiores esses valores, mais completo o modelo; e através da análise dos comportamentos exibidos pelo modelo, onde quanto mais comportamentos da descrição do sistema estiverem presentes no modelo, mais completo ele será. Logo, o modelo mais completo é que aquele que estruturalmente possui mais estados e transições e cujos estados e transições cobrem mais o comportamento definido na especificação do sistema.

Deseja-se, também, analisar a relação custo-benefício em obter-se um modelo de comportamento completo, onde custo é visto como a quantidade de casos de teste necessária para que seja extraído tal modelo. Eventualmente, o número de casos de teste pode se tornar financeira ou temporalmente impraticável, levando-se em conta a complexidade do sistema.

1.4. Organização do Texto

O texto deste trabalho está organizado da seguinte forma:

Capítulo 2 – Neste capítulo é apresentada a base teórica para entendimento dos conceitos abordados, bem como das ferramentas utilizadas ao longo do trabalho;

Capítulo 3 – Neste capítulo são especificadas todas as etapas práticas do trabalho, desde a escolha dos casos de teste, a instrumentação dos códigos, a geração de logs até a geração dos modelos de comportamento;

Capítulo 4 – Neste capítulo é realizada a análise entre o critério de cobertura de teste e o modelo de comportamento obtido para os diferentes sistemas tratados;

Capítulo 5 – Neste capítulo são apresentadas as conclusões sobre o trabalho realizado, bem como as considerações finais sobre o aprendizado adquirido com este estudo;

Apêndice – Apresenta os resultados complementares em relação aos experimentos discutidos no texto.

2. REFERENCIAL TEÓRICO

Esse capítulo apresenta os diferentes critérios de cobertura de teste escolhidos para este trabalho. Além disso, são apresentadas as ferramentas utilizadas durante a análise das coberturas de teste, bem como o processo de extração de modelos de comportamento.

2.1. Critérios de Cobertura de Teste

Os critérios de cobertura de teste definem o modo como o código será analisado estruturalmente, seja ele analisado diretamente ou através de um grafo de fluxo de controle. Entre os diferentes critérios de cobertura de teste, serão descritos a seguir: *Line Coverage*, *Branch Coverage*, *Condition Coverage*, *Path Coverage* e *Random Coverage*. Destes cinco critérios, apenas as definições de *Random Coverage* não foram baseadas em (PEZZÉ, 2008), tendo esse critério sido baseado em (DUARTE, 2007). Todos os critérios se baseiam na seleção de casos de teste (PEZZÈ, 2008). Um *caso de teste* é definido como um conjunto de condições para uma determinada execução do sistema, composto por: uma descrição do teste, resultado esperado para execução, resultado obtido a partir da execução, dados de entrada do sistema e dados de saída do sistema. Uma *suíte de teste* (PEZZÈ, 2008) é um conjunto de casos de teste para um determinado programa, sistema ou componente individual. Pode ser composta de vários casos de teste para módulos individuais, subsistemas e funcionalidades.

2.1.1. Line Coverage

O critério de cobertura por linha (*Line Coverage*) requer que cada linha do código seja exercitada por, pelo menos, um caso de teste. Formalmente, podemos definir *Line Coverage* da seguinte maneira: seja T uma suíte de teste para um programa P. T satisfaz o critério de cobertura *Line Coverage* para o programa P, se e somente se, para cada linha L em P, existe pelo menos um caso de teste em T que gera a execução de L.

A taxa do critério de cobertura por linha pode ser calculada através da proporção:

C_{Line} = número de linhas executadas / número de linhas do programa

Portanto, T satisfaz plenamente o critério de cobertura *Line Coverage* quando C_{Line} é igual a 1.

2.1.2. Branch Coverage

O critério de cobertura por desvios (*Branch Coverage*) requer que cada desvio condicional seja exercitado por, pelo menos, um caso de teste. Formalmente, podemos defini-lo da seguinte maneira: seja T uma suíte de teste para um programa P. T satisfaz o critério de cobertura *Branch Coverage* para o programa P, se e somente se, para cada desvio condicional D, existe pelo menos um caso de teste em T que gera a execução de D.

A taxa do critério de cobertura *Branch Coverage* pode ser calculada através da proporção:

C_{Branch} = número de desvios executados / número de desvios do programa

Portanto, T satisfaz plenamente o critério de cobertura *Branch Coverage* quando C_{Branch} é igual a 1. Além disso, quando obtemos C_{Branch} igual a 1, podemos inferir que C_{Line} é igual a 1. Entretanto, o contrário não é verdadeiro.

2.1.3. Condition Coverage

O critério de cobertura por condição (*Condition Coverage*) requer que cada expressão booleana básica assuma os valores *true* e *false* por, pelo menos, um caso de teste. Uma *expressão booleana básica* é definida como a menor unidade de condição, por exemplo: "*(a OR b)*" é uma expressão booleana e "*a*" e "*b*" são expressões booleanas básicas.

Formalmente, podemos defini-lo da seguinte maneira: seja T uma suíte de teste para um programa P. T satisfaz o critério de cobertura *Condition Coverage* para o programa P, se e somente se, para cada expressão booleana básica E, existe pelo menos um caso de teste em T que contém o valor *true* e pelo menos um caso de teste em T que contém o valor *false* para E.

A taxa do critério de cobertura *Condition Coverage* pode ser calculada através da proporção:

$$C_{Condition} = \frac{\text{número de valores assumidos pelas expressões booleanas básicas}}{2 * \text{número total de expressões booleanas básicas}}$$

Assim, T satisfaz plenamente o critério de cobertura *Condition Coverage* quando $C_{Condition}$ é igual a 1. Esse critério de cobertura não pode ser comparado aos supracitados, pois, apesar de testar todos os valores verdade para cada expressão booleana básica, não é possível garantir que ele cubra integralmente *Line* ou *Branch Coverage*. Isso é demonstrado através do seguinte exemplo: definimos em nosso código um comando ***if (a AND b) {...} else {...}***. Caso executemos os casos de teste (a = *true*, b = *false*) e (a = *false*, b = *true*), já terá sido atendido o critério de *Condition Coverage*, pois os valores *true* e *false* já foram atribuídos às duas variáveis. Entretanto, a avaliação da fórmula (a AND b) sempre retornou o resultado *false*, executando sempre o trecho do *else*. Portanto, o critério de cobertura por condição não garante *Line* e *Branch Coverage*.

2.1.4. Path Coverage

O critério de cobertura por caminhos (*Path Coverage*) requer que toda sequência de caminhos seja exercitada por, pelo menos, um caso de teste. Um *caminho* é definido como o fluxo percorrido no correspondente Grafo de Fluxo de Controle (CFG) a partir do nodo inicial até o nodo final. Um CFG é uma representação em forma de grafo para descrever os caminhos possíveis para um determinado sistema. Entretanto, cobrir 100% dos caminhos pode ser impossível quando tratamos de programas que possuem *loop*. Isso ocorre porque *loops* permitem que determinadas execuções atinjam tamanhos infinitos. Em virtude disto, durante a realização do trabalho, foi estipulado um tamanho

máximo para os caminhos, limitando o número de comandos de entrada para cada caso de teste. Esse tamanho máximo é definido através de um inteiro *k*, que é incluído no início do critério de cobertura. Por exemplo, *3-Path Coverage*, ou seja, esses casos de teste terão apenas 3 comandos de entrada. Além disso, o valor *k* não deve confundido com a abordagem tradicional de caminhos, onde ele é responsável por indicar a profundidade dos caminhos.

Formalmente, podemos definir o critério de cobertura *Path Coverage* da seguinte maneira: seja T uma suíte de teste para um programa P. T satisfaz o critério de cobertura *Path Coverage* para o programa P, se e somente se, para cada caminho C, existe pelo menos um caso de teste em T que exercita C.

A taxa do critério de cobertura *Path Coverage* pode ser calculada através da proporção:

C_{Path} = número de caminhos executados / número total de caminhos

Assim, T satisfaz plenamente o critério de cobertura *Path Coverage* quando C_{Path} é igual a 1. Esse critério pode ser difícil de ser satisfeito dependendo da complexidade do sistema. Entretanto, se for satisfeito, *Path Coverage* garantirá que *Line* e *Branch Coverage* são iguais a 1.

2.1.5. Random Coverage

O critério de cobertura aleatório (*Random Coverage*), utilizado na abordagem de (DUARTE, 2007) para gerar todos os rastros de execução, é baseado na geração de sequências de comandos aleatórias para um programa P. Teoricamente, não é possível garantir a plena satisfação do critério de cobertura *Random Coverage*, pois os casos de teste serão gerados de maneira aleatória, ou seja, não existe uma medida de cobertura específica. Pelo mesmo motivo, não podemos comparar este critério de cobertura com os demais critérios de cobertura de teste.

Utilizou-se esse tipo de cobertura apenas para comparar se efetivamente vale a pena seguirmos um tipo formalizado de critério de cobertura ou se é possível obter-se resultados satisfatórios seguindo uma metodologia aleatória.

2.2. Modelos de Comportamento

Para representar os modelos de comportamento durante este trabalho, foi utilizada uma representação conhecida como *Labelled Transition System* (LTS). Um LTS é definido formalmente da seguinte maneira (KELLER, 1976):

Um LTS é uma tupla $(S, A, \rightarrow, s_0)$, onde:

- S é um conjunto finito de estados;
- A é um conjunto finito de ações;
- $\rightarrow \subseteq S \times A \times S$ é uma relação de transição;
- $s_0 \in S$ é o estado inicial.

Para qualquer conjunto de ações de A, A^* é o conjunto de todas as sequências finitas de A iniciadas em s_0. Além disso, um elemento qualquer de A* é chamado de *trace* (ou, rastro) e um *trace* vazio é representado por ε. Assim, para qualquer LTS (S, A, →, s_0) com os estados s e t ∈ S e um *trace* α ∈ A^*, com α = a_0 ... a_n para n ≥ 0, nós denotamos por s $\rightarrow^{\alpha}$ t o fato de que existe s_0 ... s_n ∈ S tal que s = s_0 e t = s_n e (s_i, a_{i+1}, s_{i+1}) ∈ → para todo i, onde 0 ≤ i ≤ n.

Basicamente, um LTS descreve o comportamento discreto de determinado sistema, onde, em qualquer estado do sistema, determinadas ações podem ser realizadas, podendo levar o sistema para um novo estado. O estado inicial corresponde ao estado no qual ainda não foi realizada nenhuma ação.

2.3. Análise de Cobertura

Durante a elaboração deste trabalho, foi necessária a utilização de algumas ferramentas de apoio para realizar a análise de cobertura dos códigos estudados. Essas ferramentas automatizaram algumas etapas da análise como, por exemplo, a criação dos Grafos de Fluxo de Controle. A seguir, serão descritas as características dessas ferramentas.

2.3.1. EclEmma

EclEmma[1] é um *plugin* desenvolvido para Eclipse cuja função é analisar o código e retornar o percentual de cobertura atingido pelo código para *Line Coverage* e *Branch Coverage*, além de outras coberturas que não fazem parte do escopo do trabalho, como cobertura de classes e métodos. Além disso, a ferramenta permite, após a execução de uma suíte de testes, a realização de um *merge* dos resultados para obter a cobertura total para a suíte.

As funcionalidades do EclEmma reduziram consideravelmente o tempo de análise estrutural do código, pois foi possível alcançar 100% de *Line Coverage* e *Branch Coverage* de maneira automática, ou seja, sem a análise individual e manual dos casos de teste. Durante a utilização da ferramenta, foi verificado que o EclEmma exibia resultados de cobertura inferiores a 100% quando, na verdade, a cobertura já era integral. Essa divergência de valores ocorre devido a um problema na contabilização das instruções do código e linhas código que não são executáveis.

2.3.2. Eclipse CFG Generator

O Eclipse CFG Generator[2] é uma ferramenta desenvolvida para Eclipse e ela é responsável por gerar Grafos de Fluxo de Controle - ou simplesmente, CFGs. A ferramenta gera um CFG para cada método da classe em questão. Além de gerar os CFGs, esse *plugin* permite também a edição manual do grafo gerado, podendo-se mover

[1] http://www.eclemma.org/ Acessado em 10/08/2011.
[2] http://eclipsefcg.sourceforge.net/ Acessado em 17/08/2011.

estados, esconder estados e transições, excluir transições, etc. Um exemplo de CFG gerado está exibido na Figura 2.1.

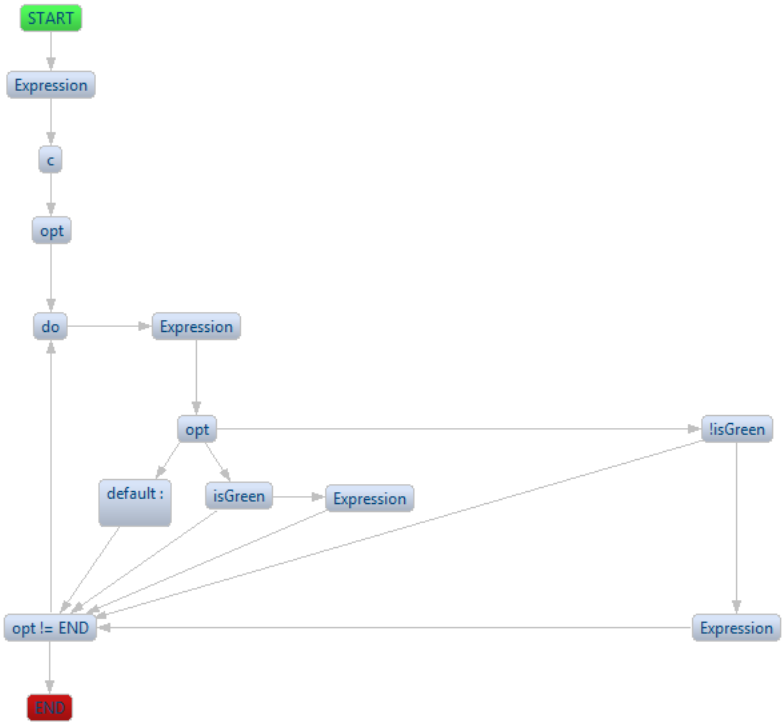

Figura 2.1: Exemplo de CFG gerado pelo Eclipse CFG Generator.

No CFG acima, *START* representa o nodo inicial do CFG; uma *Expression* representa uma atribuição de valores a um atributo ou variável do sistema; os nomes de variáveis ou atributos podem significar uma expressão booleana (*if* , *while* ou *switch*) ou uma atribuição de valores; *END* representa o nodo final do CFG; e as transições orientadas representam os caminhos possíveis para cada nodo.

Um problema encontrado nessa ferramenta é a intratabilidade de trechos de código que possuíam *try/catch*. Nesses trechos, é necessária a reescrita do código para possibilitar a geração dos CFGs. Além disso, em virtude das ambiguidades de representação e da consequente dificuldade em analisar os CFGs gerados pela ferramenta, estes foram desenhados manualmente para facilitar a compreensão dos mesmos.

2.4. Extração de Modelos

A extração de modelos é o processo de geração de um modelo de comportamento a partir de um código existente (HOLZMANN, 1999). A abordagem para extração de modelos utilizada neste trabalho segue a ideia proposta em (DUARTE, 2007) de combinar informação estática e dinâmica, ou seja, utilizamos uma estratégia híbrida baseada em análise funcional e estrutural do sistema. A seguir descreveremos as ferramentas utilizadas durante o trabalho na fase de extração de modelos de comportamento.

2.4.1. LTSE

O *Labelled Transition System Extractor*[3] (LTSE) é responsável pela extração de sistemas modelados como LTS a partir de um código Java. Esse processo de extração envolve a instrumentação do código, ou seja, incluir anotações no código original para que possamos obter as informações sobre o estado do sistema e sobre seu fluxo de controle.

Para realizarmos a instrumentação do código Java, utilizamos uma linguagem de transformação chamada TXL[4] (Turing eXtended Language). Essa linguagem recebe como parâmetros os nomes dos arquivos Java de entrada, nomes para as respectivas saídas e o caminho para as regras de instrumentação. Ela gera como saída os arquivos Java instrumentados nos arquivos de saída especificados. A execução desses códigos instrumentados gera *traces* que servem de entrada para o LTSE, o qual os converte em um modelo em FSP (Finite State Process) (MAGEE, 2006). FSP é uma álgebra de processos usada para descrever sistemas sequenciais e concorrentes que segue a semântica de um LTS.

2.4.2. LTSA

O *Labelled Transition System Analyzer*[5] (LTSA) é um verificador de modelos em LTS. Essa ferramenta compila os arquivos em FSP gerados pelo LTSE e provê ao usuário uma lista de estados e das transições do modelo de comportamento, uma visão gráfica do modelo de comportamento e ainda disponibiliza a opção de executarmos o modelo gerado. O LTSA ainda possui outras funcionalidades que estão fora do escopo deste trabalho como, por exemplo, a verificação de propriedades em lógica temporal (MANNA, 1992). A Figura 2.2 apresenta um exemplo de modelo em FSP e o respectivo código LTS gerado.

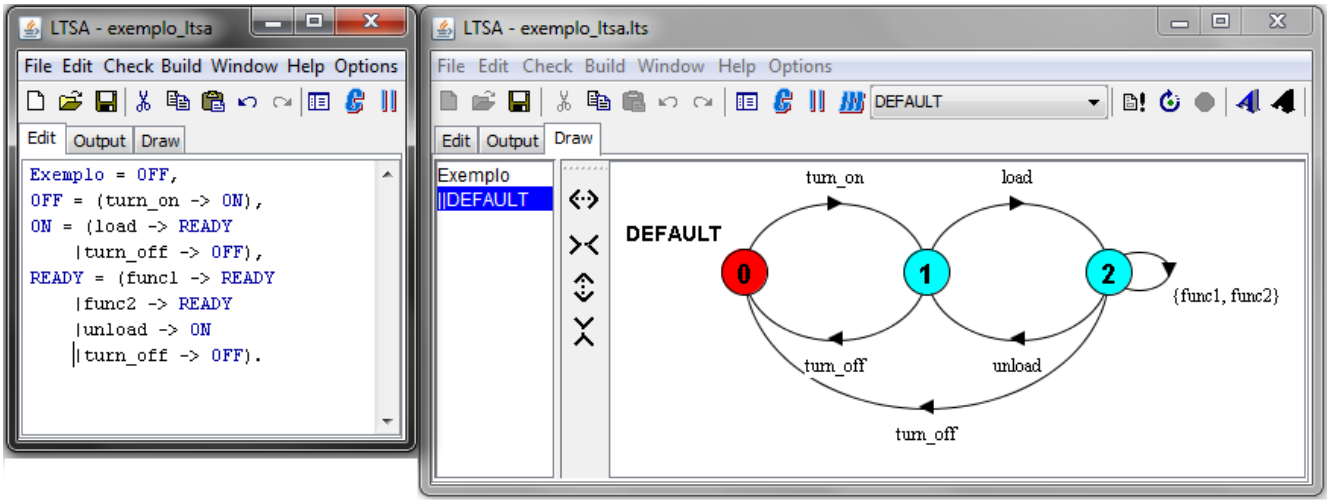

Figura 2.2: Exemplo de modelo FSP (esquerda) e o LTS gerado (direita).

[3] http://www.inf.ufrgs.br/~lmduarte/ltse/ Acessado em 20/09/2011.
[4] http://www.txl.ca/ Acessado em 20/09/2011.
[5] http://www.doc.ic.ac.uk/ltsa/ Acessado em 25/09/2011.

3. EXTRAÇÃO DE MODELOS DE COMPORTAMENTO

Neste capítulo, serão detalhadas as quatro etapas do processo de extração de modelos de comportamento: criação de casos de teste, instrumentação do código, geração de *logs* e a extração de modelos propriamente dita, de acordo com a Figura 3.1. Além disso, realizadas as quatro etapas, a seguir parte-se para uma etapa de Análise dos Modelos que, juntamente com a etapa de Geração dos Casos de Teste, é complementar ao trabalho de (DUARTE, 2007). Para descrever a metodologia utilizada, utilizamos um sistema simples para controle de semáforos de trânsito.

Figura 3.1: Diagrama de Etapas da Extração de Modelos.

3.1. Casos de Teste

Para criação da suíte de testes, é necessário haver uma análise do sistema em questão partindo de sua especificação e/ou descrição e indo até a análise estrutural do código, a fim de atender o critério que está sendo exercitado, ou seja, no caso de *Line Coverage* nosso objetivo é executar todas as linhas do código, pelo menos, uma vez. Além disso, nos casos das coberturas *Condition Coverage* e *k-Path Coverage* é necessária a análise do CFG correspondente de cada sistema. Partindo-se deste princípio, a seguir apresenta-se a descrição de um sistema de controle de semáforo de trânsito, usado como exemplo:

Descrição: O sistema é composto por um semáforo que possui duas luzes: uma vermelha e uma verde. A alternância entre as duas luzes é feita de forma manual pelo usuário através de sinais ao sistema. O sistema é iniciado pela luz vermelha e, a partir desse ponto, as luzes vermelha e verde devem se alternar até o recebimento de um comando de parada.

O código deste sistema pode ser visto na Figura 3.2:

```
1  class TrafficLights {
2      private static final int GREEN = 0;
3      private static final int RED = 1;
4      private static final int END = 2;
5      private boolean isGreen;
6
7    public TrafficLights () {
8      isGreen = false;
9      Controller c = new Controller ();
10     int opt = - 1;
11     do {
12         opt = c.nextSignal ();
13         switch (opt) {
14         case GREEN :
15             if (!isGreen)
16                 greenLights ();
17           break;
18         case RED :
19             if (isGreen)
20                 redLights ();
21           break;
22         default:
23         }
24     } while (opt != END);
25   }
26
27   private void greenLights () {
28     isGreen = true;
29     changeColour ("green");
30   }
31
32   private void redLights () {
33     isGreen = false;
34     changeColour ("red");
35   }
36
37   private void changeColour (String newColour) {
38     System.out.println (newColour);
39   }
40 }
```

Figura 3.2: Código de *TrafficLights*.

A partir da descrição, cria-se uma suíte de teste para cada diferente tipo de cobertura de testes. Como exemplo, é apresentada, na Tabela 3.1, a sequência de comandos utilizada nos casos de teste para alcançar 100% de cobertura para o critério *Condition Coverage*.

Teste ID	Entradas	Saídas
Caso de teste 01	0 2	green
Caso de teste 02	0 1 2	green red
Caso de teste 03	0 0 2	green
Caso de teste 04	0 1 1 2	green red
Caso de teste 05	f 2	

Tabela 3.1: Suíte de testes para *Condition Coverage* em *TrafficLights*.

Na Tabela 3.1, os casos de teste 01 e 02 foram criados baseados na descrição do sistema. Os demais casos de teste foram gerados a partir da análise do CFG correspondente ao código, exibido na Figura 3.3.

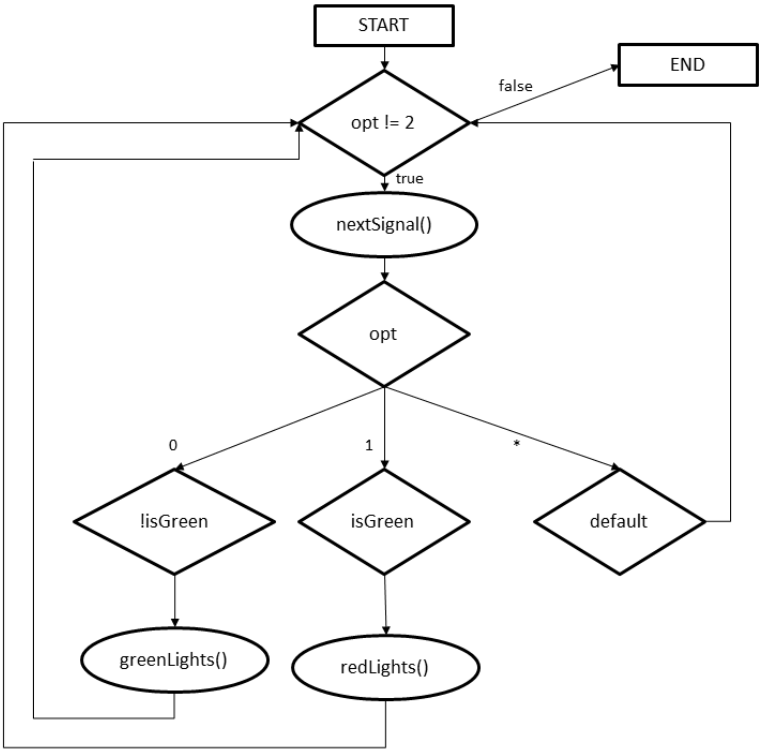

Figura 3.3: CFG de *TrafficLights*.

De modo análogo, geramos a suíte de teste para o critério *4-Path Coverage*. Para a geração dos casos de teste dos critérios *Line* e *Branch Coverage*, foi utilizada a ferramenta EclEmma para verificar a cobertura integral destes critérios de cobertura de teste. Assim, pode-se partir para a instrumentação do código.

3.2. Instrumentação

A instrumentação deve garantir que as informações necessárias ao processamento do LTSE sejam capturadas. Essas informações podem ser: estruturas de seleção, estruturas de repetição, chamadas de métodos e pontos de entrada e saída de métodos. Tais informações servem para a identificação de contextos, que são o conceito básico da abordagem proposta em (DUARTE, 2007). Como já foi mencionada anteriormente, a instrumentação do código é feita de maneira automática pela TXL. Genericamente, a instrumentação do código via TXL é realizada através do seguinte comando:

```
txl –i "<caminho_regras>" -o <outputName> <inputName> "<arquivo_de_regras>"
```

Onde: *caminho_regras* é o caminho da pasta onde estão as regras de instrumentação, *outputName* é o nome do arquivo Java de saída instrumentado, *inputName* é o nome do arquivo Java de entrada e *arquivo_de_regras* é o arquivo onde estão as regras de instrumentação providas pela TXL. Sendo assim, a instrumentação do código Java de *TrafficLights* é realizada através do seguinte comando:

```
txl –i "%INSTR_PATH%\rules" -o TrafficLightsInst TrafficLights
%INSTR_PATH%\rules\JavaInstrumentation.txl"
```

Este comando gera, em *TrafficLightsInst*, a versão instrumentada do código da Figura 3.1, segundo as regras do arquivo *JavaInstrumentation.txl*.

Opcionalmente, o LTSE provê dois arquivos *batch* para realizar a instrumentação e a desinstrumentação do código. Esses arquivos *batch* são gerados através do seguinte comando:

```
java ltse.Instrumenter <mainClass> [ -a <arguments>] -f <inputFiles> [-n <logName>]
```

Onde: <*mainClass*> é a classe principal do sistema; <*arguments*> é a lista dos argumentos necessários para execução do sistema; <*inputFiles*> é a lista dos arquivos Java a ser instrumentada; e <*logName*> é o nome do *log* a ser gerado. Em nosso caso, o comando seria o seguinte:

```
java ltse.Instrumenter TrafficLightsMain –f TrafficLights.java Controller.java
```

O código estando instrumentado, ele está pronto para gerar os *logs* que conterão a informação básica para a extração do modelo. Um trecho de código de *TrafficLightsInst* é apresentado abaixo:

```
[…]
System.err.println ("REP_ENTER:(opt != END)#" + true + "#" + getClass ().getName ()
+ "=" + hashCode () + "#" + "{" + "" + "isGreen" + "=" + isGreen + "^" + "}" + "#" + "7"
+ ";");
     {
          {
          System.err.println ("CALL_ENTER:nextSignal" + "#" + getClass ().getName ()
     + "=" + hashCode () + "#" + c + "#" + "{" + "" + "isGreen" + "=" + isGreen + "^" +
     "}" + "#" + "0" + ";");
          opt = c.nextSignal ();

          System.err.println ("CALL_END:nextSignal" + "#" + getClass ().getName () +
          "=" + hashCode () + "#" + c + "#" + "0" + ";");
[…]
private void greenLights () {
          System.err.println ("MET_ENTER:greenLights" + "#" + getClass ().getName ()
          + "=" + hashCode () + "#" + "{" + "" + "isGreen" + "=" + isGreen + "^" + "}" +
          "#" + "4" + ";");
          {
          isGreen = true;
          changeColour ("green");
          }
          System.err.println ("MET_END:greenLights" + "#" + getClass ().getName () +
          "=" + hashCode () + "#" + "4" + ";");
          }
[…]
```

No código acima, cada item em negrito visa instrumentar um comando diferente do código original de *TrafficLights*. A seguir será descrito o que cada um desses itens significa:

- REP_ENTER: Essa instrução representa a entrada em alguma estrutura de repetição. Por exemplo: *while*, *for*, etc.;
- CALL_ENTER: Essa instrução representa a chamada de um método do código;
- CALL_END: Essa instrução representa o final da execução de um método;
- MET_ENTER: Essa instrução representa o início da definição de um método;
- MET_END: Essa instrução representa o final da definição de um método.

Maiores informações sobre as regras de instrumentação podem ser encontradas em (DUARTE, 2007).

3.3. Geração de Logs

A geração de logs é a etapa onde se executam os casos de teste gerados após a análise funcional e estrutural sobre o código instrumentado. Por exemplo, a execução do "Caso de teste 01", do capítulo 3.1, é exercitada através do comandos:

```
java TrafficLightsMain 2> t1.log
> Enter next signal (0-2): 0
green
> Enter next signal (0-2): 2
```

Como "2" é o comando de parada do sistema, depois de executá-lo o arquivo *t1.log* contendo os *traces* do sistema é gerado na pasta local. O trecho do conteúdo do *log* gerado correspondente ao código instrumentado acima está apresentado abaixo:

```
REP_ENTER:(opt != END)#true#TrafficLights=9634993#{isGreen=false^}#7;
CALL_ENTER:nextSignal#TrafficLights=9634993#Controller@190d11#{isGreen=fals
e^}#0;
MET_ENTER:nextSignal#Controller=1641745#{in=java.io.BufferedReader@a90653^}
#1;
MET_END:nextSignal#Controller=1641745#1;
CALL_END:nextSignal#TrafficLights=9634993#Controller@190d11#0;
[…]
REP_END:(opt != END)#TrafficLights=9634993#7;
[…]
END
```

3.4. Extração do Modelo de Comportamento

Tendo em mãos todos os *logs* para um determinado critério de cobertura de testes, pode-se partir para a geração do modelo de comportamento. Nesta etapa, deve-se, primeiramente, gerar o arquivo de refinamento, que define os atributos que o LTSE utilizará para extrair o modelo. No caso de *TrafficLights*, o único atributo presente no sistema é a variável booleana *isGreen*. Neste caso, cria-se um arquivo chamado "TF.ref" que conterá apenas o nome do atributo. Criado o arquivo de refinamento, o seguinte comando gera o modelo de comportamento:

```
java ltse.LTSExtractor TF.ref  t1.log  t2.log  t3.log  t4.log  t5.log
```

Após a execução do comando acima, o LTSE cria uma série de arquivos, na pasta local, com os seguintes formatos: *.ctb, *.lts, *.mdl e *.trc. A seguir, detalha-se o que contém cada um dos arquivos gerados:

- CTB: Esse arquivo contém a tabela de contextos do código (DUARTE, 2008);
- LTS: Esse arquivo contém os dados a serem analisados pela ferramenta LTSA. Através dele, podemos visualizar o LTS correspondente ao modelo de comportamento e simular a execução do mesmo;
- MDL: Contém os dados do modelo propriamente dito.

A seguir, com auxílio do LTSA, pode-se visualizar e analisar o modelo de comportamento gerado. A Figura 3.4 apresenta o modelo gerado pelo critério de cobertura *Condition Coverage*.

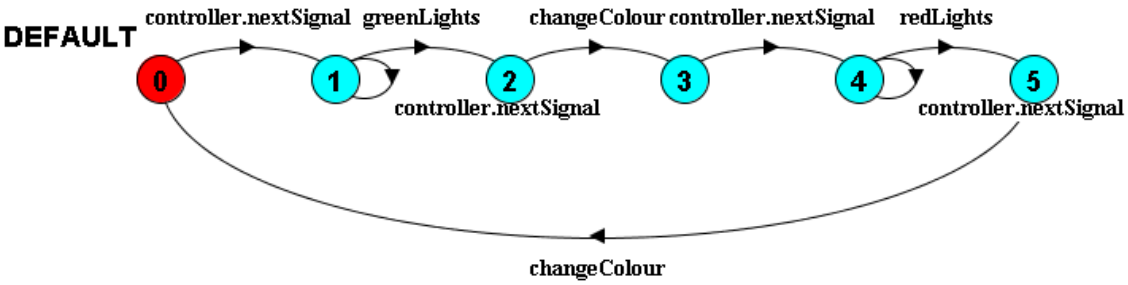

Figura 3.4: Modelo de *TrafficLights* para a cobertura *Condition Coverage*.

Na Figura 3.4, o estado "0" é o estado inicial do modelo de comportamento. Além disso, cada transição rotulada representa a execução de um método do código e os *loops* dos estados "1" e "4" representam a entrada de um comando inválido.

3.5. Outros Experimentos

Além do sistema *TrafficLights*, foi realizado o trabalho da extração de modelos de comportamento para outros três sistemas que serão descritos abaixo:

- **ACController**: O *ACController* é um sistema para gerenciar o funcionamento do ar condicionado de um determinado ambiente. Esse sistema é baseado em dois sensores: um termostato e um sensor que controla a porta do ambiente (se está aberta ou fechada). O sistema permite que o ar condicionado seja ligado apenas quando a porta do ambiente estiver fechada e a temperatura acima de um limiar. Quando a porta do ambiente for aberta e o ar condicionado estiver ligado, ele deverá ser imediatamente desligado. Além disso, o ar condicionado não pode estar ligado se a temperatura do ambiente estiver abaixo do limiar;

- **Editor**: O *Editor* é um sistema para edição de documentos que permite a abertura, edição, impressão, gravação e o fechamento do Editor. Nenhuma função do sistema é habilitada enquanto algum documento não for aberto. Um documento só pode ser salvo, se ele sofre uma ação de edição. Ao tentar encerrar o *Editor*, caso o programa tenha sido editado e não tenha sido salvo, o programa questionará o usuário se deseja salvar o documento ou não;

- **Bounded Buffer**: O *Bounded Buffer* é uma aplicação concorrente tradicional onde é simulada a relação produtor-consumidor com um *buffer* de tamanho limitado igual a 3. Essa aplicação é dividida em três componentes: *Buffer*, *Producer* e *Consumer*. Neste caso, foi realizada a extração de modelos de comportamento para os critérios *Line Coverage* e *Branch Coverage* apenas para demonstrar que o processo de extração de modelos de comportamento pode ser expandido para aplicações concorrentes. Além disso, há a questão da dificuldade em reproduzir os resultados obtidos porque os sistemas são concorrentes e podem gerar resultados diferentes para os mesmos parâmetros de entrada.

A seguir, são exibidos todos os resultados obtidos para o estudo de caso do *ACController*. Os itens apresentados são código Java, casos de teste para cada critério de teste para as cinco diferentes coberturas de teste e o correspondente modelo de comportamento extraído:

```
class AirConditioner implements Signals {
  private static boolean room_hot;
  private static boolean door_closed;
  private static boolean ac_on;
  public AirConditioner (EnvController c) {
    room_hot = false;
    door_closed = true;
    ac_on = false;
    boolean finished = false;
    int message = - 1;
    while (! finished) {
        message = c.nextSignal ();
      switch (message) {
        case ROOM_HOT :
            if (! room_hot) {
                room_hot = true;
             System.out.println ("-> Room hot");
            if (door_closed) {
                ac_on = true;
               System.out.println ("-> AC on");
            }
            }
            break;
        case ROOM_COOL :
            if (room_hot) {
                room_hot = false;
            System.out.println ("-> Room cool");
            if (ac_on) {
                ac_on = false;
              System.out.println ("-> AC off");
            }
            }
          break;
        case DOOR_OPEN :
            if (door_closed) {
                door_closed = false;
            System.out.println ("-> Door open");
            if (ac_on) {
                ac_on = false;
              System.out.println ("-> AC off");
            }
            }
          break;
        case DOOR_CLOSED :
            if (! door_closed) {
                door_closed = true;
            System.out.println ("->Door closed");
            if (room_hot) {
                ac_on = true;
              System.out.println ("-> AC on");
            }
            }
            break;

        case OFF :
                finished = true;
            break;
        default :
                System.out.println ("Incorrect command!");
      }
    }
  }
}
```

Figura 3.5: Código de *ACController*.

Nos modelos apresentados a seguir, renomeamos o método *nextSignal* para *nS* a fim de facilitar a visualização dos modelos.

Teste ID	Entradas
Caso de teste 01	0 2 4
Caso de teste 02	0 1 4
Caso de teste 03	0 4
Caso de teste 04	2 0 3 4
Caso de teste 05	f 4

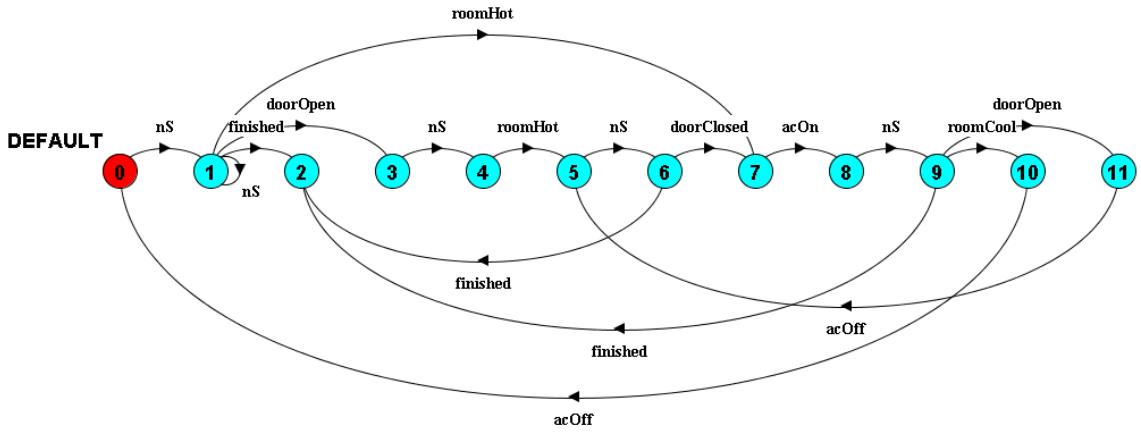

Figura 3.6: Casos de teste e modelo de *ACController* para *Line Coverage*.

Teste ID	Entradas
Caso de teste 01	0 2 4
Caso de teste 02	0 1 4
Caso de teste 03	0 4
Caso de teste 04	2 0 3 4
Caso de teste 05	f 4

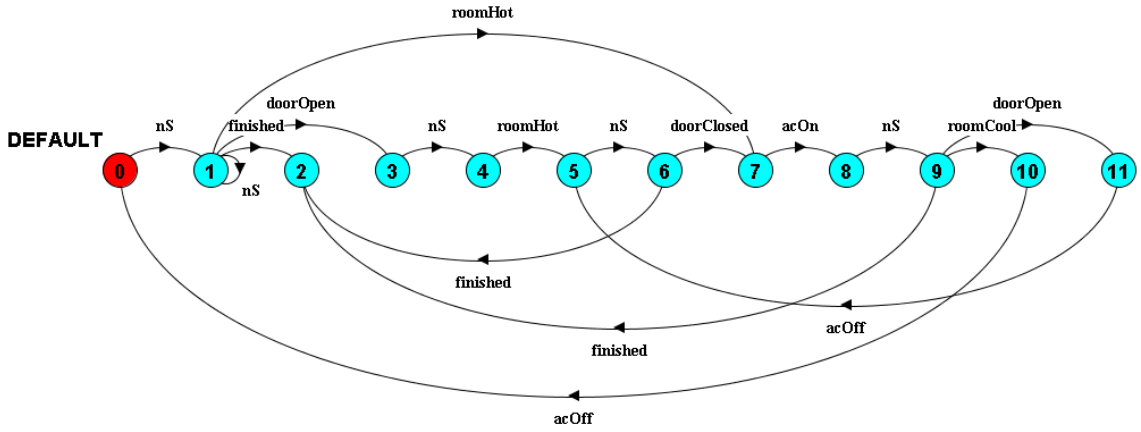

Figura 3.7: Casos de teste e modelo de *ACController* para *Branch Coverage*.

Teste ID	Entradas	Teste ID	Entradas
Caso de teste 01	2 4	Caso de teste 07	0 f 4
Caso de teste 02	f 4	Caso de teste 08	2 f 4
Caso de teste 03	0 4	Caso de teste 09	0 2 4
Caso de teste 04	1 4	Caso de teste 10	0 3 4
Caso de teste 05	2 3 4	Caso de teste 11	0 1 4
Caso de teste 06	2 0 4	Caso de teste 12	3 4

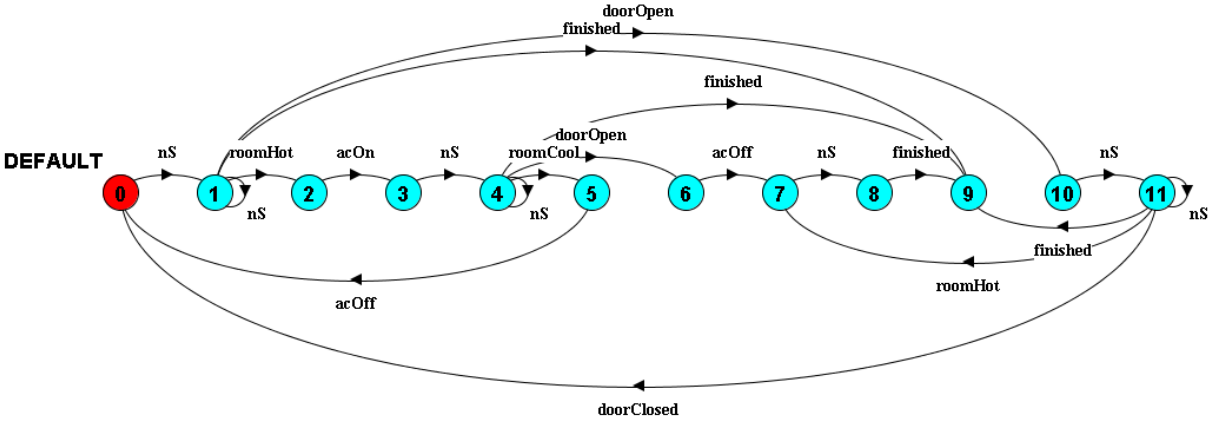

Figura 3.8: Casos de teste e modelo de *ACController* para *Condition Coverage*.

Teste ID	Entradas
Caso de teste 01	2 0 1 4
Caso de teste 02	0 0 1 1 4
Caso de teste 03	4
Caso de teste 04	2 2 0 0 4
Caso de teste 05	2 2 1 3 0 f 3 1 0 4
Caso de teste 06	0 1 4
Caso de teste 07	f 3 4

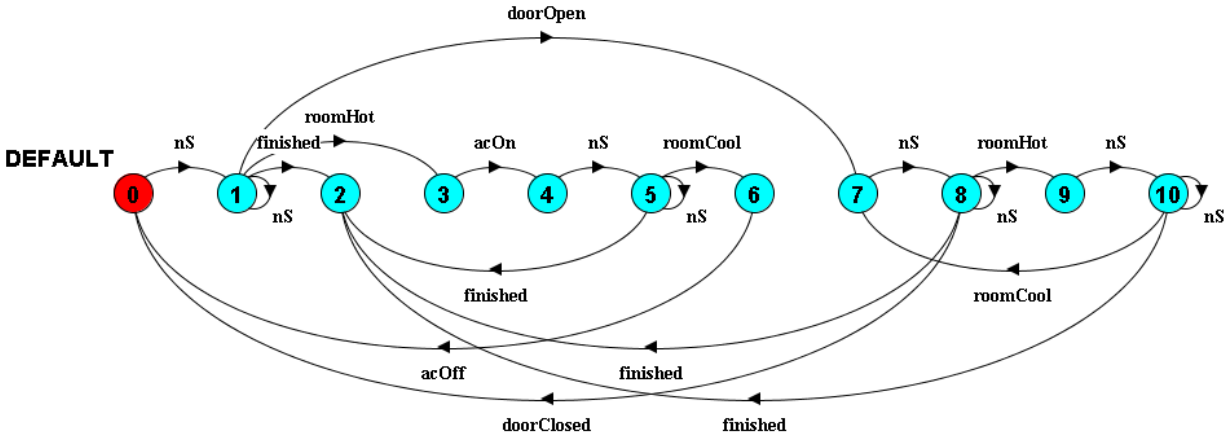

Figura 3.9: Casos de teste e modelo de *ACController* para *Random Coverage*.

Teste ID	Entradas	Teste ID	Entradas	Teste ID	Entradas
Caso de teste 01	0 0 4	Caso de teste 10	1 f 4	Caso de teste 19	3 3 4
Caso de teste 02	0 1 4	Caso de teste 11	2 0 4	Caso de teste 20	3 f 4
Caso de teste 03	0 2 4	Caso de teste 12	2 1 4	Caso de teste 21	f 0 4
Caso de teste 04	0 3 4	Caso de teste 13	2 2 4	Caso de teste 22	f 1 4
Caso de teste 05	0 f 4	Caso de teste 14	2 3 4	Caso de teste 23	f 2 4
Caso de teste 06	1 0 4	Caso de teste 15	2 f 4	Caso de teste 24	f 3 4
Caso de teste 07	1 1 4	Caso de teste 16	3 0 4	Caso de teste 25	f f 4
Caso de teste 08	1 2 4	Caso de teste 17	3 1 4	Caso de teste 26	4
Caso de teste 09	1 3 4	Caso de teste 18	3 2 4		

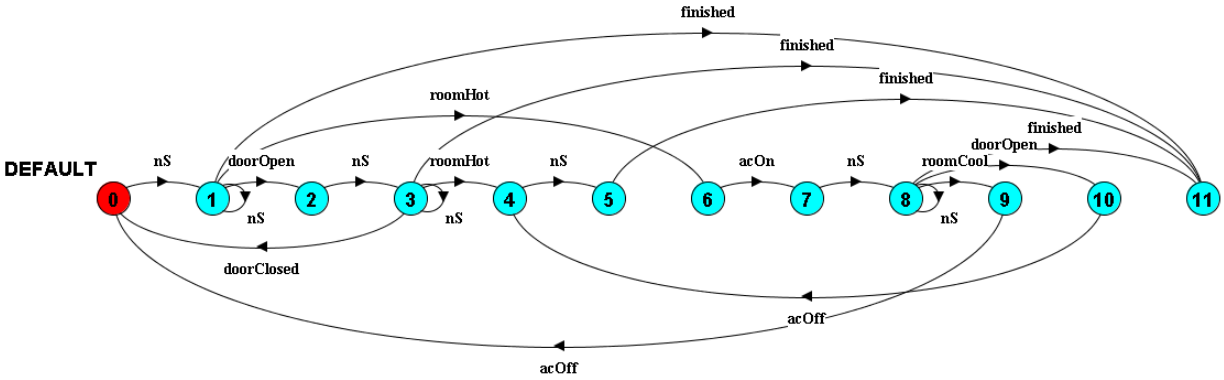

Figura 3.10: Casos de teste e modelo de *ACController* para *3-Path Coverage*.

As demais informações referentes a todos os sistemas deste trabalho estão disponíveis no Apêndice.

4. ANÁLISE DE COBERTURA DE TESTE X MODELO DE COMPORTAMENTO

Neste capítulo, realizaremos a análise dos modelos de comportamento gerados em relação aos diferentes critérios de cobertura utilizando uma análise estrutural e uma análise observacional.

4.1. Critérios

Como forma para definir qual ou quais coberturas de teste geraram os melhores modelos para cada um dos sistemas, utilizamos dois tipos de critérios distintos: critério estrutural e critério observacional.

4.1.1. Critério Estrutural

Para análise utilizando o critério estrutural, utilizamos o número de estados e o número de transições do modelo de comportamento gerado pelo LTSE para avaliar os modelos extraídos. Esses valores são disponibilizados pela ferramenta LTSA.

Lembrando que neste nível de abstração, todo o modelo gerado pela ferramenta é considerado correto, ou seja, nenhuma transição ou estado é inválido. Assim, o critério estrutural aponta os modelos de comportamento mais completos, ou seja, quais modelos de comportamento possuem mais estados e transições válidos para cada sistema.

No caso de *TrafficLights*, *Line Coverage* e *Branch Coverage* obtiveram um modelo de comportamento menos completo em comparação às demais coberturas de teste. Os resultados para cada diferente critério de cobertura está apresentado na Tabela 4.1:

Sistema	Cobertura	Refinamento	Nº Estados	Nº Transições	Nº Casos de Teste
TrafficLights	Line	isGreen	6	7	3
TrafficLights	Branch	isGreen	6	7	3
TrafficLights	Condition	isGreen	6	8	5
TrafficLights	Aleatório	isGreen	6	8	4
TrafficLights	4-Path	isGreen	6	8	27

Tabela 4.1: Resultados para *TrafficLights*.

No caso do *Editor*, *4-Path Coverage* obteve o modelo de comportamento mais completo para o sistema entre as diferentes coberturas de teste. Os demais critérios apresentaram a ausência de alguns estados e transições do sistema. Os resultados para cada diferente critério de cobertura está apresentado na Tabela 4.2:

Sistema	Cobertura	Refinamento	Nº Estados	Nº Transições	Nº Casos de Teste
Editor	Line	isSaved, isOpen	10	14	6
Editor	Branch	isSaved, isOpen	10	14	6
Editor	Condition	isSaved, isOpen	10	18	10
Editor	Aleatório	isSaved, isOpen	10	19	8
Editor	4-Path	isSaved, isOpen	12	22	34

Tabela 4.2: Resultados para *Editor*.

No caso do *ACController*, *3-Path Coverage* obteve o modelo de comportamento mais completo para o sistema entre as diferentes coberturas de teste. Os demais critérios apresentaram a ausência de alguns estados e transições do sistema. Os resultados para cada diferente critério de cobertura está apresentado na Tabela 4.3:

Sistema	Cobertura	Refinamento	Nº Estados	Nº Transições	Nº Casos de Teste
ACConditioner	Line	ac_on, door_closed, room_hot	12	18	5
ACConditioner	Branch	ac_on, door_closed, room_hot	12	18	5
ACConditioner	Condition	ac_on, door_closed, room_hot	12	21	12
ACConditioner	Aleatório	ac_on, door_closed, room_hot	7	13	7
ACConditioner	3-Path	ac_on, door_closed, room_hot	12	21	25

Tabela 4.3: Resultados para *ACController*.

No caso de *Bounded Buffer*, *Line Coverage* e *Branch Coverage* alcançaram o modelo completo. A composição de seus componentes gerou os resultados apresentados na Tabela 4.4:

Sistema	Cobertura	Nº Estados	Nº Transições	Nº Casos de Teste
Bounded Buffer	Line	144	401	6
Bounded Buffer	Branch	144	401	6

Tabela 4.4: Resultados para *Bounded Buffer*.

4.1.2. Critério Observacional

Para análise utilizando o critério observacional, faremos uma análise das funcionalidades de cada sistema para ver, independentemente do critério estrutural, quais coberturas geraram um modelo de comportamento mais completo em relação à sua descrição. Para isso, foram analisados os comportamentos contidos na descrição e foi verificado se eles eram contemplados no modelo de comportamento através da execução dos mesmos na ferramenta LTSA.

No caso de *TrafficLights*, apenas os critérios *Line Coverage* e *Branch Coverage* não atenderam completamente à especificação. Nestes dois critérios, o sistema não permite comandos inválidos quando o atributo *isGreen* é *true*.

No caso do *Editor*, apenas *4-Path Coverage* atendeu completamente à descrição do sistema. A seguir serão citadas as funcionalidades que não foram atendidas em cada uma das demais coberturas:

- *Branch Coverage*: O modelo de comportamento gerado não permite que o Editor seja fechado antes de abrir um documento e depois que o documento é editado e ainda não foi salvo, ele pode apenas Salvar o documento ou Sair do Editor – podendo salvar ou não após o comando de saída. Além disso, não são permitidos comandos incorretos após a edição do documento;
- *Line Coverage*: O modelo de comportamento gerado não permite que o Editor seja fechado antes de abrir um documento e depois que o documento é editado e ainda não foi salvo, ele pode apenas Salvar o documento ou Sair do Editor – podendo salvar ou não após o comando de saída. Além disso, não são permitidos comandos incorretos após a edição do documento;
- *Condition Coverage*: O modelo de comportamento gerado não permite que o Editor seja fechado antes de abrir um documento e depois que o documento é editado e ainda não foi salvo, ele pode apenas Salvar o documento ou Sair do Editor - podendo salvar ou não após o comando de saída;
- *Random Coverage*: O modelo de comportamento gerado não permite apenas que o documento seja editado múltiplas vezes, sem antes o documento ser salvo;

No caso do *ACController*, nenhum dos critérios de cobertura conseguiu atender integralmente à descrição do sistema. A seguir serão citadas as funcionalidades que não foram atendidas em cada uma das coberturas:

- *Branch Coverage*: O modelo de comportamento gerado não permite que o ar condicionado seja desligado antes do ambiente ficar com temperatura agradável, também não é possível fechar a porta imediatamente depois que ela é aberta ou imediatamente após o ambiente esquentar;
- *Line Coverage*: O modelo de comportamento gerado não permite que o ar condicionado seja desligado antes do ambiente ficar com temperatura agradável, também não é possível fechar a porta imediatamente depois que ela é aberta ou imediatamente após o ambiente esquentar;
- *Condition Coverage*: O modelo de comportamento gerado não permite que o ar condicionado seja religado imediatamente após desligá-lo e não permite que o ar condicionado seja desligado antes do ambiente ficar com temperatura agradável. Além disso, se a porta do ambiente é aberta enquanto o ar condicionado está ligado, o sistema permite apenas que o ar condicionado seja desligado e posteriormente o sistema seja fechado;
- *Random Coverage*: O modelo de comportamento gerado não permite que a porta do ambiente seja aberta quando o ar condicionado está ligado e não permite a abertura da porta quando o ambiente está quente;
- *3-Path Coverage*: O modelo de comportamento gerado não permite que o ar condicionado seja religado imediatamente após desligá-lo. Além disso, se a porta do ambiente é aberta enquanto o ar condicionado está ligado, o sistema permite apenas que o ar condicionado seja desligado e posteriormente o sistema seja fechado.

4.2. Discussão de Resultados

Neste capítulo são discutidos os resultados obtidos ao longo do trabalho e apresentados na seção anterior. Além disso, é discutido o impacto de cada critério de cobertura para cada diferente sistema, juntamente com uma análise geral de cada critério de cobertura e sua relação custo-benefício.

4.2.1. Coberturas de Teste

Como pode ser observado nos resultados apresentados, as coberturas de *Branch* e *Line Coverage* só geraram modelos de comportamento completos para sistemas de complexidade muito baixa, assumindo a posição de piores critérios de cobertura. No caso de *Random Coverage*, não podemos garantir que ele irá atender a completude mesmo para sistemas muito simples. Entretanto, a chance de alcançar a completude aumenta proporcionalmente de acordo com o número de casos de teste.

No caso de *Condition Coverage*, apesar de só ter gerado o modelo de comportamento mais completo para *TrafficLights*, é um dos critérios de cobertura com maior capacidade de alcançar a completude dos modelos, ficando atrás apenas de *Path Coverage*. No caso de *Path Coverage*, apesar do número de casos de teste ser relativamente superior às demais coberturas de teste, é a melhor alternativa para alcançar o modelo de comportamento mais completo possível dados os critérios de cobertura analisados.

4.2.2. Relação Custo-Benefício

Apesar de *Path Coverage* ser a melhor alternativa entre os critérios de cobertura de teste, cobrir todos os caminhos para sistemas que contenham atributos com domínio de valores consideravelmente grandes, em vez de, por exemplo, atributos com valores booleanos (*true* e *false*), ou sistemas muito complexos pode tornar o número de casos de testes muito grande ou, até mesmo, impraticável. Por exemplo, em um sistema com 10 alternativas de comandos de entrada, para alcançarmos 100% da cobertura para *3-Path Coverage*, seriam necessários 1000 casos de teste. Nesses casos, é necessário verificar a necessidade de obter-se um modelo de comportamento completo ou se uma aproximação seria suficiente como, por exemplo, para analisar uma determinada propriedade do sistema.

4.2.3. Análise dos Sistemas Estudados

4.2.3.1. TrafficLights

Em virtude da simplicidade de tal sistema, os modelos de comportamento gerados por *Random Coverage*, *Condition Coverage* e *4-Path Coverage* conseguiram atender integralmente à descrição do sistema.

Em relação à *Line Coverage* e *Branch Coverage*, apesar de termos atingido 100% dessas coberturas, o mesmo comportamento não pode ser verificado nesses casos. Isso ocorreu, pois a parte do código referente aos comandos inválidos foi exercitada quando o atributo *isGreen* era *false*, ou seja, os *traces* não possuíam a informação necessária para aceitar comandos inválidos quando o atributo era *true*.

4.2.3.2. Editor

No caso do *Editor*, como havia sido citado anteriormente, apenas *4-Path Coverage* atendeu integralmente à descrição do sistema. Além disso, verificando que todos os critérios de cobertura de teste, com exceção de *Random Coverage*, atingiram 100% fica evidente que apenas atender integralmente o critério de cobertura não garante um modelo de comportamento completo.

4.2.3.3. ACController

No caso do *ACController*, ocorreu a mesma situação que no Editor, ou seja, com exceção de *Random Coverage*, todos os critérios de cobertura atingiram 100%, mas nenhum dos modelos extraídos atendeu integralmente à descrição do sistema.

Neste caso, a única maneira de alcançar o modelo de comportamento completo seria aumentar o tamanho de *Path Coverage* para *4-Path Coverage*, ao invés de *3-Path Coverage*. Entretanto, ao invés de 26 (5^2+1) casos de teste, seriam necessários 126 (5^3+1) casos de teste para que a funcionalidade ausente fosse exercitada. Em virtude disso, a questão sobre a relação custo-benefício deve ser analisada nesse caso.

4.2.3.4. Bounded Buffer

No caso de *Bounded Buffer*, todos os componentes (*Buffer*, *Consumer* e *Producer*), alcançaram seus modelos individuais completos. Entretanto, a análise completa da composição dos componentes, em virtude do alto número de estados e transições, ficou a cargo do verificador LTSA.

4.3. Trabalhos Correlatos

O presente trabalho está relacionado a duas abordagens distintas de Extração de Modelos de Comportamento. A seguir, será descrita cada uma dessas abordagens e a diferença entre a metodologia utilizada na composição deste:

- (COOK, 1998) e (LORENZOLI, 2006): Esses trabalhos realizam a Extração de Modelos a partir de *Traces* utilizando uma abordagem de inferência de gramáticas para tentar identificar dependências entre eventos do sistema com base em relações estatísticas presentes no conjunto de rastros de execução. Entretanto, não é pré-requisito básico a utilização de Testes para geração dos *traces*. Assim, nossa abordagem se difere ao realizar uma etapa de Criação de Casos de Teste e também pela posterior etapa de análise dos Modelos de Comportamento gerados e os Critérios de Cobertura de Teste utilizados para extração de cada modelo;

- (GROZ, 2008) e (WALKINSHAW, 2009): Esses trabalhos, diferente dos anteriores, se aproximam mais de nossa abordagem, pois utilizam Testes para a composição dos *traces* e, posteriormente, usam o modelo para gerar novos testes, cujos resultados são depois adicionados ao modelo anterior. Logo, eles não se preocupam com o conjunto inicial de testes porque vão incrementalmente aumentando a completude dos modelos. Entretanto, não há garantia de se atingir a completude total e nem há qualquer análise de como o conjunto inicial de testes afeta o processo como um todo.

5. CONCLUSÃO

O estudo e o extenso exercício do processo da extração de modelos de comportamento, passando pela análise estática do código, criação e análise dos CFGs, criação da suíte de testes, instrumentação dos códigos, geração dos *traces* e a geração dos modelos de comportamento, permitiram a coleta de grande quantidade de resultados que possibilitaram a análise consistente a respeito da influência da escolha do critério de cobertura de teste sobre o modelo de comportamento obtido. Além disso, comprovou-se a capacidade da abordagem utilizada em tratar e gerar modelos de comportamento para aplicações concorrentes.

Através da etapa de análise dos resultados, foi possível determinar que o critério *k-Path Coverage*, entre os critérios de cobertura estudados, é o critério com maior potencial em obter o modelo de comportamento mais completo. Além disso, pode-se verificar que optar por um determinado critério de cobertura de testes não está relacionado apenas ao critério que provê os melhores resultados. Entretanto, deve-se atentar ao custo temporal e/ou financeiro para realização de tal processo para sistemas de alta complexidade.

Como citado no início do trabalho, foram utilizadas algumas ferramentas para automatizar etapas do processo da extração de modelos. Entretanto, ainda há etapas que são obrigatoriamente realizadas de forma manual, ou seja, estão mais propensas a apresentar erros provenientes da habilidade do analisador/testador. Como exemplo de etapas que apresentam essa desvantagem, pode-se citar a análise dos CFGs e a análise do modelo de comportamento extraído em relação a sua especificação e ao seu respectivo código. Além disso, a ferramenta LTSA apresenta uma limitação do número de estados para apresentação do modelo LTS gerado. Por exemplo, no caso de estudo do *Bounded Buffer*, não foi possível visualizar o modelo LTS do sistema que possui 144 estados.

Por fim, este trabalho poderia servir como base para futuros estudos que poderiam visar à diversificação dos critérios de cobertura de teste utilizados neste trabalho ou, até mesmo, a união de diferentes critérios a fim de obter melhores modelos de comportamento. Além disso, o trabalho poderia ser realizado utilizando-se sistemas de complexidade mais alta. Do mesmo modo, esse trabalho poderia ser expandido também para aplicações reais onde, teoricamente, a qualidade dos modelos de comportamento deve ser mantida, respeitando os critérios de cobertura adotados neste trabalho.

6. REFERÊNCIAS

CLARKE, L. A.; Grumberg, O., Peled, D. A., **"Model Checking"**, The MIT Press, Cambridge, MA, USA, 1999.

COOK, J. E. & Wolf, A. L., **"Discovering Models of Software Processes from Event-Based Data"**, ACM Transactions on Software Engineering and Methodology 7(3), p. 215-249, 1998.

DUARTE, L. M., **"Behaviour Model Extraction using Context Information"**, PhD thesis, Imperial College London, University of London, 2007.

DUARTE, L. M.; Kramer, J. & Uchitel, S., **"Towards Faithful Model Extraction Based on Contexts"**, Lecture Notes in Computer Science, 4961, Proceedings of FASE 2008, Springer, Budapest, Hungary, p. 101-115, 2008.

GROZ, R.; Li, K.; Petrenko, A. & Shahbaz, M., **"Modular System Verification by Inference, Testing and Reachability Analysis"**, in 'Proceedings of the 20th IFIP TC 6/WG 6.1 international conference on Testing of Software and Communicating Systems: 8th International Workshop', Springer-Verlag, Berlin, Heidelberg, p. 216-233, 2008.

HOLZMANN, G. & Smith, M., **"A Practical Method for Verifying Event-Driven Software"**, in 'International Conference on Software Engineering', ACM New York, Los Angeles, USA, p. 597-607, 1999.

KELLER, R., **"Formal Verification of Parallel Programs"**, Communications of the ACM 19(7), p. 371-384, 1976.

LORENZOLI, D.; Mariani, L. & Pezzè, M., **"Inferring State-Based Behavior Models"**, in 'WODA '06: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis', ACM Press, New York, NY, USA, p. 25-32, 2006.

MAGEE, J. & Kramer, J., **"Concurrency: State Models and Java Programming"**, Wiley and Sons, 2006.

MANNA, Z. & Pnueli, A., **"The Temporal Logic of Reactive and Concurrent Systems"**, Springer-Verlag New York, Inc., New York, NY, USA,1992.

PEZZÈ, M., Young, M., **"Software Testing and Analysis"**, Wiley, 2008.

WALKINSHAW, N.; Derrick, J. & Guo, Q., **"Iterative Refinement of Reverse-Engineered Models by Model-Based Testing"**, in 'Proceedings of the 2nd World Congress on Formal Methods', Springer-Verlag, Berlin, Heidelberg, p. 305-320, 2009.

APÊNDICE

A seguir serão apresentados os casos de teste e os resultados para *TrafficLights* que não foram apresentados ao longo do trabalho:

Teste ID	Entradas
Caso de teste 01	0 2
Caso de teste 02	0 1 2
Caso de teste 03	f 2

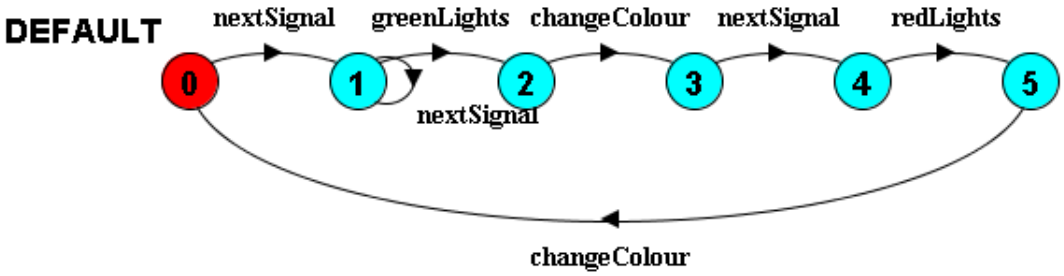

Figura 7.1: Casos de teste e modelo de *TrafficLights* para *Line Coverage*.

Teste ID	Entradas
Caso de teste 01	0 2
Caso de teste 02	0 1 2
Caso de teste 03	f 2

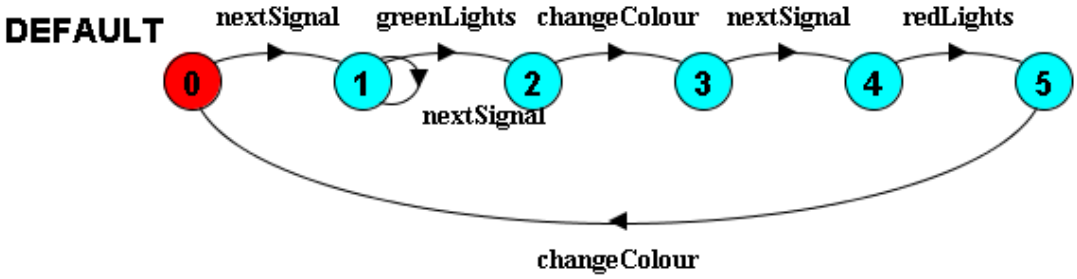

Figura 7.2: Casos de teste e modelo de *TrafficLights* para *Branch Coverage*.

Teste ID	Entradas
Caso de teste 01	1 0 1 2
Caso de teste 02	2
Caso de teste 03	0 0 2
Caso de teste 04	1 0 2

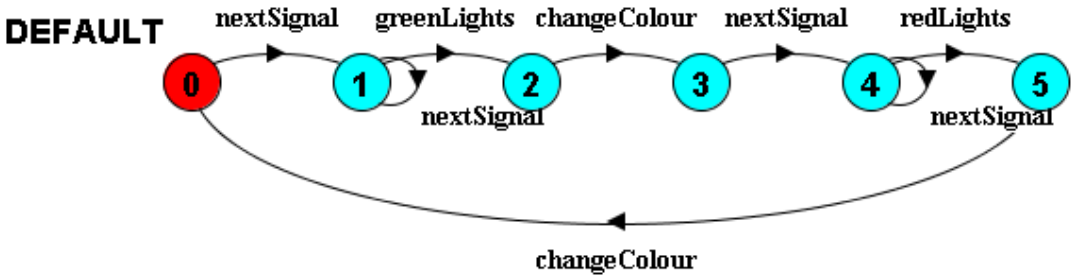

Figura 7.3: Casos de teste e modelo de *TrafficLights* para *Random Coverage*.

Teste ID	Entradas	Teste ID	Entradas	Teste ID	Entradas
Caso de teste 01	0 0 0 2	Caso de teste 11	f 0 0 2	Caso de teste 21	0 1 f 2
Caso de teste 02	0 0 1 2	Caso de teste 12	0 f f 2	Caso de teste 22	0 f 1 2
Caso de teste 03	0 1 1 2	Caso de teste 13	f 0 f 2	Caso de teste 23	1 0 f 2
Caso de teste 04	0 1 0 2	Caso de teste 14	f f 0 2	Caso de teste 24	1 f 0 2
Caso de teste 05	1 0 0 2	Caso de teste 15	1 f 1 2	Caso de teste 25	f 0 1 2
Caso de teste 06	1 0 1 2	Caso de teste 16	1 1 f 2	Caso de teste 26	f 1 0 2
Caso de teste 07	1 1 0 2	Caso de teste 17	f 1 1 2	Caso de teste 27	f f f 2
Caso de teste 08	1 1 1 2	Caso de teste 18	1 f f 2	Caso de teste 28	2
Caso de teste 09	0 0 f 2	Caso de teste 19	f 1 f 2		
Caso de teste 10	0 f 0 2	Caso de teste 20	f f 1 2		

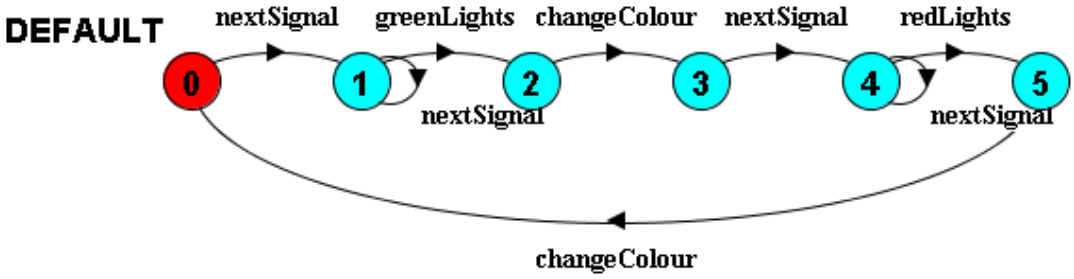

Figura 7.4: Casos de teste e modelo de *TrafficLights* para *4-Path Coverage*.

A seguir serão apresentados o código, os casos de teste e os resultados para *Editor* que não foram apresentados ao longo do trabalho:

```
class Editor {
  private boolean isOpen;
  private boolean isSaved;
  private CommandReader r;
  public Editor (CommandReader cr) {
    isOpen = false;
    isSaved = true;
    r = cr;
    int cmd = - 1;
    while (cmd != 4) {
        try {
        System.out.print ("\n> Enter command: ");
        String rc;
        rc = r.readCommand ();
        cmd = Integer.parseInt (rc);
       } catch (Exception e) {}
      switch (cmd) {
        case 0 :
            if (! isOpen)
                open ();
          break;
        case 1 :
            if (isOpen)
                edit ();
          break;
       case 2 :
            if (isOpen)
                print ();
          break;
        case 3 :
            if (! isSaved)
                save ();
            break;
        case 4 :
            exit ();
          break;
      }
    }
  }
 void open () {
    isOpen = true;
    System.out.println ("> File opened");
  }
  void edit () {
    isSaved = false;
      System.out.println ("> File modified");
  }
  void print () {
    System.out.println ("> File printed");
  }
  void save () {
    isSaved = true;
    System.out.println ("> File saved");
  }
  void close () {
    isOpen = false;
    System.out.println ("> File closed");
  }
  void exit () {
        if (!isSaved) {
            try {
        System.out.print ("> Save modifications? ");
        int opt = 0;
        String rc;
        rc = r.readCommand ();
        opt = Integer.parseInt (rc);
        if (opt == 0)
            save ();
      } catch (Exception e) {}
        }
    if (isOpen)
        close ();
  }
}
```

Figura 7.5: Código de *Editor*.

Teste ID	Entradas
Caso de teste 01	0 4
Caso de teste 02	0 1 4 0
Caso de teste 03	0 2 4
Caso de teste 04	0 1 4 5
Caso de teste 05	0 1 3 4
Caso de teste 06	0 f 4

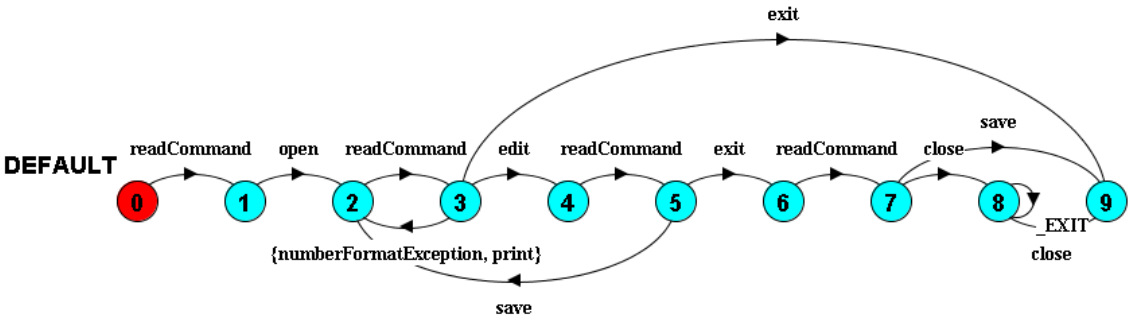

Figura 7.6: Casos de teste e modelo de *Editor* para *Line Coverage*.

Teste ID	Entradas
Caso de teste 01	0 4
Caso de teste 02	0 1 4 0
Caso de teste 03	0 2 4
Caso de teste 04	0 1 4 5
Caso de teste 05	0 1 3 4
Caso de teste 06	0 f 4

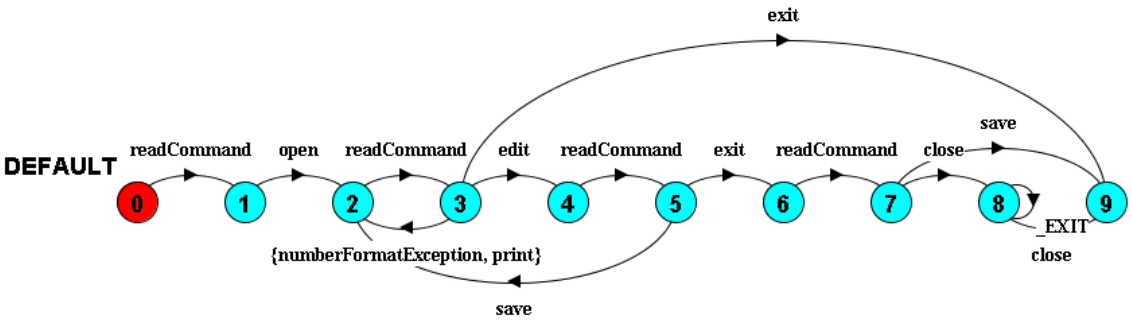

Figura 7.7: Casos de teste e modelo de *Editor* para *Branch Coverage*.

Teste ID	Entradas
Caso de teste 01	0 4
Caso de teste 02	f 4
Caso de teste 03	0 1 4 5
Caso de teste 04	0 1 3 4
Caso de teste 05	2 4
Caso de teste 06	3 4
Caso de teste 07	0 2 4
Caso de teste 08	1 4
Caso de teste 09	0 1 4 0
Caso de teste 10	0 0 4

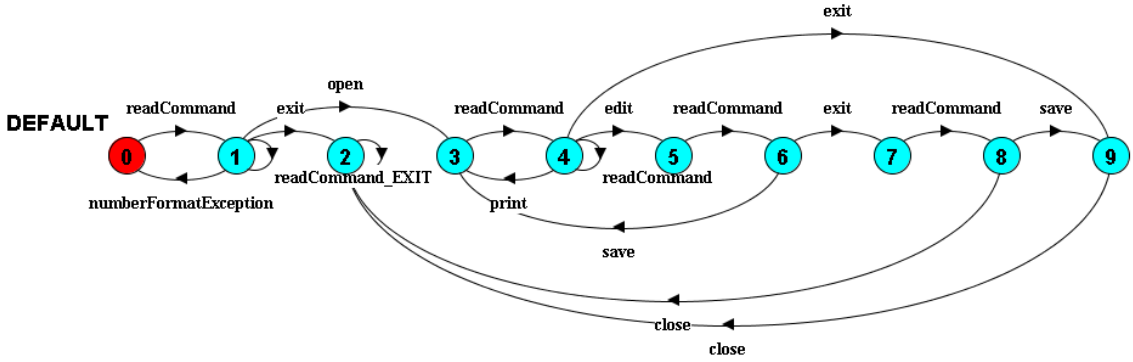

Figura 7.8: Casos de teste e modelo de *Editor* para *Condition Coverage*.

Teste ID	Entradas
Caso de teste 01	0 3 4
Caso de teste 02	2 3 4
Caso de teste 03	2 3 0 3 3 0 2 4
Caso de teste 04	4
Caso de teste 05	1 0 0 1 3 4
Caso de teste 06	0 2 4
Caso de teste 07	1 2 1 2 0 1 2 4 f
Caso de teste 08	2 3 3 0 3 3 1 0 4

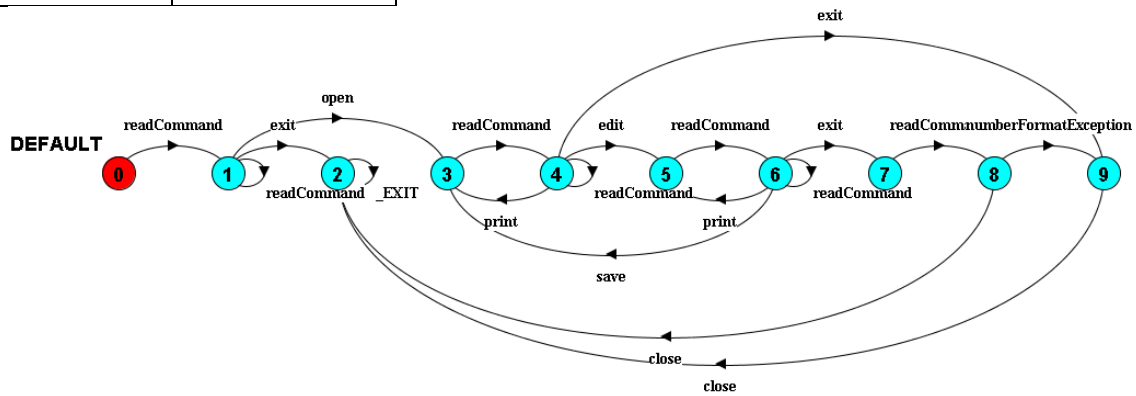

Figura 7.9: Casos de teste e modelo de *Editor* para *Random Coverage*.

Teste ID	Entradas	Teste ID	Entradas	Teste ID	Entradas
Caso de teste 01	0 0 1 4 0	Caso de teste 13	0 1 f 4 0	Caso de teste 25	0 3 3 4
Caso de teste 02	0 0 1 4 5	Caso de teste 14	0 1 f 4 5	Caso de teste 26	0 3 f 4
Caso de teste 03	0 0 2 4	Caso de teste 15	0 2 0 4	Caso de teste 27	0 f 0 4
Caso de teste 04	0 0 3 4	Caso de teste 16	0 2 1 4 0	Caso de teste 28	0 f 1 4 0
Caso de teste 05	0 0 f 4	Caso de teste 17	0 2 1 4 5	Caso de teste 29	0 f 1 4 5
Caso de teste 06	0 1 0 4 0	Caso de teste 18	0 2 2 4	Caso de teste 30	0 f 2 4
Caso de teste 07	0 1 0 4 5	Caso de teste 19	0 2 3 4	Caso de teste 31	0 f 3 4
Caso de teste 08	0 1 1 4 0	Caso de teste 20	0 2 f 4	Caso de teste 32	0 f f 4
Caso de teste 09	0 1 1 4 5	Caso de teste 21	0 3 0 4	Caso de teste 33	0 0 0 4
Caso de teste 10	0 1 2 4 0	Caso de teste 22	0 3 1 4 0	Caso de teste 34	4
Caso de teste 11	0 1 2 4 5	Caso de teste 23	0 3 1 4 5		
Caso de teste 12	0 1 3 4	Caso de teste 24	0 3 2 4		

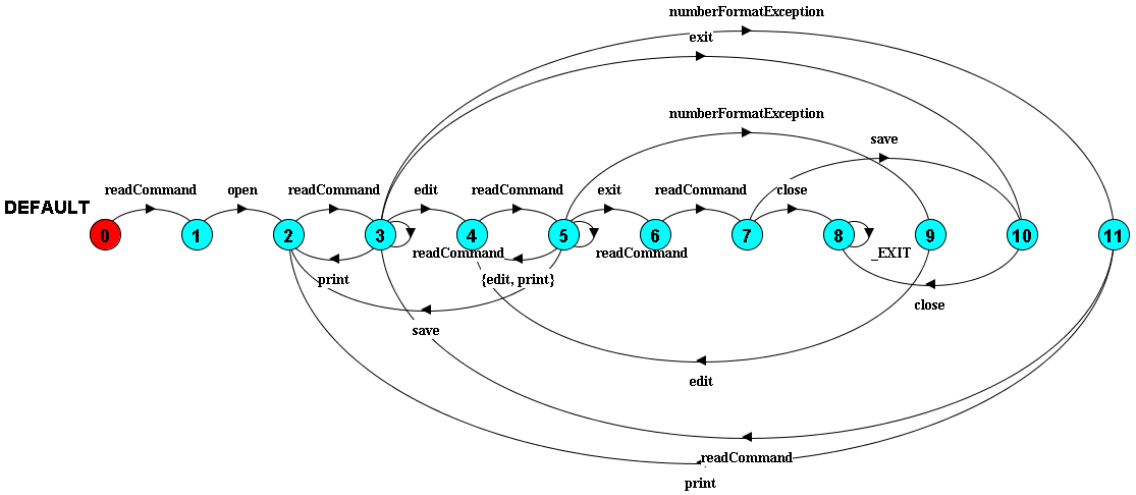

Figura 7.10: Casos de teste e modelo de *Editor* para *4-Path Coverage*.

A seguir serão apresentados o código e os casos de teste utilizados para extração do modelo de *Bounded Buffer* para os critérios de *Line* e *Branch Coverage*. Os modelos não serão inclusos porque o LTSA não tem suporte gráfico para exibição de modelos dessa magnitude.

```
1  class Buffer implements BufferInterface {
2      static final int SIZE = 3;
3      protected Object[] array = new Object[SIZE];
4      protected int putPtr = 0;
5      protected int getPtr = 0;
6      protected int usedSlots = 0;
7      protected boolean halted = false;
8      public synchronized void put (Object x, int id) {
9          while (usedSlots == SIZE) {
10             try {
11                 System.out.println ("Producer "+id+" waits");
12                 wait ();
13             }
14             catch (InterruptedException ex) {   }
15         }
16
17         int data = ((Attribute) x).attr;
18         System.out.println ("Producer "+id+": buf[" + putPtr + "] = " + data);
19
20         array[putPtr] = x;
21         putPtr = (putPtr + 1) % SIZE;
22         if (usedSlots == 0)
23             notifyAll ();
24         usedSlots++;
25     }
26     public synchronized Object get (int id) throws HaltException {
27         while (usedSlots == 0 && !halted) {
28             try {
29                 System.out.println ("Consumer "+id+" waits");
30                 wait ();
31             }
32             catch (InterruptedException ex) {   }
33         }
34         if (usedSlots == 0) {
35             System.out.println ("Consumer "+id+" gets halt exception");
36             HaltException he = new HaltException ();
37             throw (he);
38         }
39         Object x = array[getPtr];
40         int data = ((Attribute) x).attr;
41         System.out.println ("Consumer "+id+": "+ data + " = buf[" + getPtr + "]");
42         getPtr = (getPtr + 1) % SIZE;
43         if (usedSlots == SIZE)
44             notifyAll ();
45         usedSlots--;
46         return x;
47     }
48     public synchronized void halt (int id) {
49         System.out.println ("Producer "+id+" sets halt flag");
50         halted = true;
51         notifyAll ();
52     }
53 }
```

Figura 7.11: Código do componente *Buffer* de *Bounded Buffer*.

```
class Consumer extends Thread {
    private int id;
    protected int count;
    private int n_values;
    private Buffer buffer;
    public Consumer (int i, Buffer b, int n) {
        id = i;
    buffer = b;
    n_values = n;
    System.out.println ("Consumer "+id+" started");
    this.start ();
  }
    public void run () {
        AttrData[] received = new AttrData[10];
        count = 0;
        boolean exception = false;
        try {
            while (count < n_values) {
                received[count] = (AttrData) buffer.get (id);

                count++;
            }
        }
        catch (HaltException e) {
            System.out.println("Exception caught: " + e.getClass().getSimpleName());
            exception = true;
        }
        System.out.println ("Consumer "+id+" stopped");
    }
}
```

Figura 7.12: Código do componente *Consumer* de *Bounded Buffer*.

```
class Producer extends Thread {
    private int id;
    protected int count;
    private int n_values;
  private Buffer buffer;
  public Producer (int i, Buffer b, int n) {
    id = i;
    buffer = b;
    n_values = n;
    System.out.println ("Producer "+id+" started");
    this.start ();
  }
  public void run () {
    count = 0;
    for (int i = 0; i < n_values; i ++) {
            AttrData ad = new AttrData (i, i * i);
      buffer.put (ad, id);
      count++;
      yield ();
    }
        buffer.halt (id);
        System.out.println ("Producer "+id+" stopped");
    }
}
```

Figura 7.13: Código do componente *Producer* de *Bounded Buffer*.

Teste ID	Entradas
Caso de teste 01	10 50 50
Caso de teste 02	30 50 50
Caso de teste 03	50 50 50
Caso de teste 04	50 30 50
Caso de teste 05	50 50 30
Caso de teste 06	50 50 70

Tabela 7.1: Casos de teste de *Bounded Buffer* para *Line* e *Branch Coverage*.